版权注意事项：
1、书籍版权归作者和出版社所有
2、本PDF仅限用于个人获取知识，进行私底下的知识交流
3、PDF获得者不得在互联网上以任何目的进行传播
4、如觉得书籍内容很赞，请购买正版实体书，支持作者
5、请于下载PDF后24小时内删除本PDF。

TURING 图灵程序设计丛书

위키북스

【韩】李成旭 著
武传海 译



深入理解MariaDB与MySQL

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

李成旭

计算机工程专业，曾担任金融企业CRM/DW项目负责人，用两年多时间推进针对DW的ETL解决方案的开发。担任NHN的DBA时，负责标准化与数据建模、DB运营和技术标杆管理。目前在Kakao公司（韩国即时通信软件运营商）负责MariaDB的升级和构建所需的代码安全及标杆管理工作。著有《PHP 5 Web编程》《MySQL性能优化》《MySQL权威指南》。

武传海

擅韩语，喜计算机，有多年翻译经验，内容涉及多个领域，尤其擅长翻译各类计算机图书，已出版多部韩语译著。

QQ：768160125
jeonhae@126.com

封面设计 @CHOCDESIGN

TURING 图灵程序设计丛书



深入理解MariaDB与MySQL

【韩】李成旭 著
武传海 译

人民邮电出版社
北京

图书在版编目（CIP）数据

深入理解MariaDB与MySQL /（韩）李成旭著；武传海译. -- 北京：人民邮电出版社，2015.12
（图灵程序设计丛书）
ISBN 978-7-115-40908-9

Ⅰ. ①深… Ⅱ. ①李… ②武… Ⅲ. ①关系数据库系统 Ⅳ. ①TP311.138

中国版本图书馆CIP数据核字（2015）第267480号

内 容 提 要

本书讲解了MariaDB的诞生背景及设计目的，并通过与MySQL的比较使读者轻松理解MariaDB。本书内容涉及MariaDB 5.1～10.0的各个方面，讲解重点在于MariaDB 10.0的各种实用功能。此外，MariaDB与MySQL有着紧密的联系，书中对MariaDB 10.0与MySQL 5.6的新功能进行了比较，读者可以同时学习MariaDB与MySQL的内容。

◆ 著　　[韩]李成旭
译　　武传海
责任编辑　傅志红
执行编辑　陈　曦
责任印制　杨林杰
◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京鑫正大印刷有限公司印刷
◆ 开本：800×1000　1/16
印张：28.5
字数：622千字　　2015年12月第1版
印数：1-3 000册　　2015年12月北京第1次印刷
著作权合同登记号　图字：01-2014-6026号

定价：89.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京崇工商广字第0021号

推荐语

为了提升 MariaDB 性能，MariaDB 基金会承诺引入优化器等存储引擎和各种改进手段、管理功能。

另外，我们也会持续发展 MariaDB 社区，并通过开源倡导者和许多其他人的参与，继续达成 MariaDB（其基础也在不断发展）开源的目标。

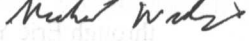
围绕着 MariaDB 基金会，许多全球性的 IT 公司如淘宝和谷歌等，也已加入 MariaDB 这个热闹的大家庭。我坚信，未来还会有很多公司期望参与 MariaDB 生态系统的建设。

我希望在韩国有更多人加入 MariaDB 社区，也有更多公司开始使用 MariaDB。

我个人感谢并祝贺李成旭在韩国出版了第一本关于 MariaDB 的书，并希望更多 DBA 从本书中体会到 MariaDB 的特性和功能之美。

MariaDB 基金会

CTO Michael "Monty" Widenius



MariaDB 首席宣传官

Colin Charles



前言

我从 2010 年左右开始写作《MySQL 权威指南》一书，与其他商用 RDBMS 相比，MySQL 服务器的用途相当明确。使用 MySQL 服务器的环境中，即使部分数据发生损坏或者服务器突然崩溃，也不会有什么大问题。因此，许多数据库管理员对 MySQL 抱着“MySQL 服务器就应该这样啊！”的看法。MySQL 服务器大量应用于无专门 DBA 的初创企业，通常由忙于业务开发的人进行管理。而且 MySQL 服务器通常运行在硬件与配置不是非常好的环境中，这就导致在此类环境中使用过 MySQL 服务器的大部分用户会对其有先入为主的偏见，认为它是一个“非常奇怪的数据库”。

《深入理解 MariaDB 与 MySQL》原书出版于 2014 年 4 月，现在，MySQL 不再是那种不稳定的免费数据库了，也不只在小规模的初创企业或门户网站中使用。目前，连支付系统都在使用 MySQL 服务器，原来只使用 MSSQL 或 Oracle DBMS 的游戏服务也逐渐开始使用 MySQL 服务器。不仅是小规模的移动游戏，甚至 MMORPG 这样的大型游戏也表现出使用 MySQL 服务器的趋势。Oracle 或 MSSQL 服务器昂贵的许可价格在客观上不断壮大着 MySQL 服务器的用户队伍，但最重要的原因还在于 MySQL 服务器本身的可靠性以及超高的性价比，这是毋容置疑的。希望本书能为 MySQL 的进一步普及尽一份绵薄之力。

不知不觉，《MySQL 权威指南》已经出版三年多了。在此期间，MySQL 市场也开始呈现多样化，其中最具代表性的是 PerconaServer 以及本书要讲解的 MariaDB。PerconaServer 从属于 Oracle 发布的 MySQL 服务器系列，但 MariaDB 在功能或设计哲学上表现出相当独立的发展方向，相信这种趋势会越来越明显。

目前，谷歌与中国最大的网上商城——“淘宝”开始为 MariaDB 贡献源代码。对这些企业巨头而言，MariaDB 的先天限制就是其优点。MariaDB 是在 Oracle MySQL 源代码基础上开发而成的，因而不能像 Oracle 一样轻易地商业化。也就是说，它不可能像 MySQL 企业版一样，将源代码封闭起来以发布商业付费版本。对谷歌和淘宝这类需要修改 MySQL 源代码并使用的企业而言，MySQL 社区版只有部分功能可用，远不如 MariaDB 更有吸引力。这类技术实力雄厚的大型企业选用 MariaDB 不仅意味着其是值得信赖的，而且他们开发并改善的各种功能可以免费提供给每个用户。

本书面向对 MariaDB 与 MySQL 感兴趣的读者，讲解时采用比较说明的方式，对 MariaDB 10.0 与 MySQL 5.6 的个性和共性进行介绍。通过阅读本书，各位应当能够理解 MariaDB 与 MySQL 的特点以及二者之间的功能差异，并能根据实际提供的服务选择合适的解决方案。

目录

第1章 MariaDB 1

1.1 MariaDB 2

1.2 MariaDB 与 MySQL 3

1.2.1 MariaDB、MySQL 和 PerconaServer 3
1.2.2 相同点 6
1.2.3 不同点 7
1.2.4 各版本间的兼容性 10
1.2.5 性能比较 11
1.2.6 MariaDB 是否会取代 MySQL 14
1.2.7 MariaDB 与 MySQL 的选择 14

第2章 安装 17

2.1 下载 18

2.2 安装 20

2.2.1 在 Windows 操作系统下安装 MariaDB 21
2.2.2 在 Linux 操作系统下安装 MariaDB 27

2.3 更新 28

2.3.1 从 MySQL 升级到 MariaDB 时的注意事项 28
2.3.2 不受版本限制的安全升级法 29
2.3.3 从 MySQL 5.0 或之前版本升级为 MariaDB 5.5 30
2.3.4 从 MySQL 5.1 升级为 MariaDB 5.5 31
2.3.5 从 MySQL 5.5 升级为 MariaDB 5.5 32

第3章 MariaDB启动及执行查询 33

3.1 启动与终止服务器 MariaDB MySQL 34

3.1.1 在 Linux 中启动与终止 MariaDB 服务器 34
3.1.2 在 Windows 系统中启动与终止 MariaDB 服务器 36

3.2 登录服务器 MariaDB MySQL 37
3.2.1 登录服务器与版本确认 37
3.2.2 mysql 客户端程序支持的选项 39
3.3 创建数据库与用户 MariaDB MySQL 41
3.3.1 MariaDB 用户账户识别与权限 41
3.3.2 MariaDB 默认用户 47
3.3.3 MariaDB 的默认数据库 48
3.3.4 新建数据库 50
3.3.5 创建用户 50
3.4 创建与修改数据表 MariaDB MySQL 51
3.4.1 创建数据表 52
3.4.2 修改数据表（在线与离线） 53
3.4.3 删除数据表 66
3.5 数据操作 MariaDB MySQL 67
3.5.1 INSERT 67
3.5.2 SELECT 68
3.5.3 UPDATE 70
3.5.4 REPLACE 71
3.5.5 DELETE 71

第4章 执行计划分析 73

4.1 概要 MariaDB MySQL 74
4.1.1 查询执行过程 74
4.1.2 优化器种类 75
4.1.3 统计信息 76
4.1.4 直方图统计信息 MariaDB 82
4.1.5 连接优化器选项 87
4.2 准备示例数据 92
4.2.1 加载示例数据 92
4.2.2 收集统计信息 94
4.3 执行计划分析 MariaDB MySQL 96
4.3.1 id 列 97
4.3.2 select_type 列 98
4.3.3 table 列 106

4.3.4 type 列 108
4.3.5 possible_keys 列 117
4.3.6 key 列 117
4.3.7 key_len 列 117
4.3.8 ref 列 119
4.3.9 rows 列 120
4.3.10 Extra 列 121
4.3.11 EXPLAIN EXTENDED（Filtered 列） 149
4.3.12 EXPLAIN EXTENDED（附加优化器信息） 150
4.3.13 EXPLAIN PARTITIONS（Partitions 列） 151

4.4 优化器提示 MariaDB MySQL 152

4.4.1 提示的使用方法 152
4.4.2 STRAIGHT_JOIN 153
4.4.3 USE INDEX/FORCE INDEX/IGNORE INDEX 156
4.4.4 SQL_CACHE/SQL_NO_CACHE 157
4.4.5 SQL_CALC_FOUND_ROWS 158
4.4.6 其他提示 160

4.5 分析执行计划时的注意事项 160

4.5.1 Select_type 列中需要注意的项目 160
4.5.2 Type 列中需要注意的项目 161
4.5.3 Key 列中需要注意的项目 161
4.5.4 Rows 列中需要注意的项目 161
4.5.5 Extra 列中需要注意的项目 161

第5章 优化 163

5.1 全表扫描 MariaDB MySQL 164

5.2 ORDER BY 处理（Using filesort） MariaDB MySQL 165

5.2.1 排序缓冲 165
5.2.2 排序算法 167
5.2.3 排序处理方式 169
5.2.4 ORDER BY..LIMIT n 优化 MySQL 177
5.2.5 与排序相关的状态变量 178

5.3 GROUP BY 处理 MariaDB MySQL 179

5.3.1 使用索引扫描处理 GROUP BY（紧凑索引扫描） 179
5.3.2 使用松散索引扫描处理 GROUP BY 179
5.3.3 使用临时表处理 GROUP BY 181

5.4 DISTINCT 处理 MariaDB MySQL 182
5.4.1 SELECT DISTINCT... 182
5.4.2 DISTINCT 用于集合函数内部 183
5.5 临时表（Using temporary） MariaDB MySQL 184
5.5.1 需要使用临时表的查询 185
5.5.2 在磁盘上创建临时表（使用 Aria 存储引擎） MariaDB 185
5.5.3 与临时表有关的状态变量 186
5.5.4 带索引的内部临时表 187
5.5.5 内部临时表的注意事项 187
5.6 索引条件下推 MariaDB MySQL 188
5.7 多范围读 MariaDB MySQL 191
5.7.1 基于 RowId 排序（Rowid-orderd scan） 192
5.7.2 基于 Key 排序（Key-ordered scan） MariaDB 194
5.7.3 基于 Key 与 RowId 排序（Key-ordered、Rowid-ordered scan） MariaDB 195
5.7.4 “多范围读”优化与排序 197
5.7.5 “多范围读”优化的注意事项 198
5.8 索引合并 MariaDB MySQL 199
5.8.1 Using union 201
5.8.2 Using sort_union 202
5.8.3 Using intersect 204
5.8.4 Using sort_intersect MariaDB 205
5.9 数据表连接 MariaDB MySQL 205
5.9.1 连接类型 206
5.9.2 连接算法 212
5.9.3 连接的注意事项 222
5.10 子查询 MariaDB MySQL 224
5.10.1 半连接子查询优化 224
5.10.2 非半连接的子查询优化 236
5.10.3 子查询缓存 238

第6章 存储引擎 241
6.1 Aria 存储引擎 MariaDB 242
6.1.1 事务 242

6.1.2 页面缓存 243
6.1.3 系统变量设置 243

6.2 XtraDB 存储引擎 MariaDB 244

6.3 InnoDB 存储引擎 MariaDB MySQL 247

6.3.1 MySQL 5.6 InnoDB 247
6.3.2 “脏页”刷新 262
6.3.3 缓冲池性能改善 MariaDB 279
6.3.4 原子写（针对 FusionIO SSD 的 Atomic write） MariaDB 283
6.3.5 扩展的 InnoDB 引擎状态输出 MariaDB 284
6.3.6 XtraDB 重做日志存档 288
6.3.7 跟踪修改页面 MariaDB 289

6.4 全文检索引擎 MariaDB MySQL 290

6.4.1 添加全文检索索引 290
6.4.2 全文检索索引表空间 292
6.4.3 与全文检索索引相关的 INFORMATION_SCHEMA 信息 294
6.4.4 使用全文检索索引 295
6.4.5 注意事项 295

6.5 Memcached 插件 MySQL 298

6.5.1 架构 298
6.5.2 安装与测试 299
6.5.3 缓存策略 303
6.5.4 注册用户数据表 304
6.5.5 相关系统变量 305

6.6 Cassandra 存储引擎 MariaDB 306

6.6.1 Cassandra 307
6.6.2 Cassandra 存储引擎 309

6.7 CONNECT 存储引擎 MariaDB 311

6.7.1 安装 CONNECT 存储引擎 311
6.7.2 连接 Oracle RDBMS 数据表 312
6.7.3 连接 my.cnf 设置文件 313
6.7.4 连接操作系统目录 314

6.8 Sequence 存储引擎 MariaDB 315

6.8.1 Sequence 存储引擎的基本用法 315
6.8.2 查找遗漏的编号 317
6.8.3 创建顺序组合的号码对 318

6.8.4 查找倍数或公倍数 318
6.8.5 顺序生成字母表 319
6.8.6 顺序生成日期 320
6.8.7 数据复制加工 320

6.9 Mroonga 全文检索存储引擎 MariaDB MySQL 321
6.9.1 索引算法 322
6.9.2 安装 Mroonga 全文检索引擎 324
6.9.3 使用 Mroonga 全文检索引擎 325

第7章 其他功能 329

7.1 性能提升 MariaDB MySQL Enterprise 330

7.2 管理与诊断 MariaDB 335
7.2.1 SHOW EXPLAIN FOR<THREAD-ID> 335
7.2.2 向慢查询日志输出执行计划 337
7.2.3 输出结构化的执行计划 338
7.2.4 线程的内存使用量 340
7.2.5 SHUTDOWN 命令 341
7.2.6 强制终止用户或查询执行 341
7.2.7 GET DIAGNOSTICS 343

7.3 开发效率 MariaDB 344
7.3.1 LIMIT ROWS EXAMINED 344
7.3.2 DELETE...RETURNING... 346
7.3.3 以微秒为单位保存时间 347
7.3.4 设置 DATETIME 类型的默认值 349
7.3.5 扩展正则表达式功能 349
7.3.6 虚拟列 352
7.3.7 动态列 354

7.4 分区 MariaDB MySQL 358
7.4.1 显式指定分区 358
7.4.2 交换分区表空间 362

7.5 备份 MariaDB MySQL 364
7.5.1 二进制日志远程备份 365
7.5.2 XtraBackup 介绍 368
7.5.3 XtraBackup 的功能 381

7.5.4 XtraBackup 的高级用法 386

第8章 复制 397

8.1 全局事务 ID MariaDB MySQL 398

8.1.1 全局事务 ID 的定义 398
8.1.2 全局事务 ID 的必要性 400
8.1.3 MariaDB 10.0 全局事务 ID MariaDB 403
8.1.4 MySQL 5.6 全局事务 ID MySQL 413

8.2 多源复制 MariaDB 421

8.2.1 多源复制相关命令 422
8.2.2 构建多源复制 423
8.2.3 多源复制与全局事务 426

8.3 多线程复制 MariaDB MySQL 428

8.3.1 MySQL 5.6 的多线程复制 428
8.3.2 MariaDB 10.0 的多线程复制 431

8.4 Crash safe slave MariaDB MySQL 433

8.4.1 MariaDB 10.0 的崩溃安全复制 434
8.4.2 MySQL 5.6 的崩溃安全复制 435

8.5 改善基于 ROW 的复制功能 MariaDB MySQL 435

8.5.1 ROW 格式的容量优化 MySQL 435
8.5.2 ROW 格式二进制日志的信息性日志事件 436

8.6 延迟复制 MySQL 439

8.7 MariaDB 与 MySQL 服务器间的复制 440

8.8 其他功能改善 MariaDB MySQL 441

8.8.1 二进制日志校验和 441
8.8.2 二进制日志 API 442
8.8.3 提交二进制日志组 443

第1章

MariaDB

1.1 MariaDB
1.2 MariaDB与MySQL

1.1 MariaDB

MariaDB 是 Monty Program AB 公司推出的一款独立的 RDBMS，类似于 MySQL 与 Oracle 数据库管理系统。注意，此处的“独立”并不是指它可以通过插件方式使用 InnoDB 等存储引擎。事实上，很难用三言两语讲清楚 MariaDB。下面看看 MariaDB 数据库的几个主要特征，通过这些特征，各位可以大致了解 MariaDB 数据库。

1. MariaDB 诞生于 MySQL Community 代码数据库。

MariaDB 是在 MySQL（开源的 Community 版本）代码基础上创建而成的 RDBMS。MariaDB 各个版本分别基于 MySQL Community 的对应版本，后面会详细说明。但是，MariaDB 并不是原封不动地使用 MySQL Community 版本代码，Monty Program AB 公司在保证它与 MySQL Community 版本拥有相同功能的前提下，添加了新功能。由于 MariaDB 建立在 MySQL Community 版本之上，因而两者之间具有良好的兼容性，在很多方面是相通的。

2. MariaDB 是由 Monty Program AB 维护的 MySQL。

Monty Program AB 是 MySQL 创始人 Michael “Monty” Widenius（下称 “Monty”）2009 年创建的公司。MySQL 先由 Sun Microsystems 收购，后来又落入 Oracle 之手。MySQL 的许可证政策也随之发生变化，开发人员遭到大量替换。据说，许多 MySQL 的核心开发人员转投到 Monty Program AB 公司工作。Oracle 发布 MySQL Community 新版本时，Monty Program AB 的 MySQL 专家都会改善既有代码，添加额外功能，发布 MariaDB 新版本。因此，可以将 MariaDB 视为 MySQL Community 相应版本的增强版本。

3. MariaDB 是开源数据库。

与以前一样，Oracle MySQL Community 版本仍然是开源的。许多用户都有这样的疑问：“既然 MySQL Community 版本是开源的，那为什么还要推出另外一个开源的 RDBMS 呢?” 事实上，Oracle MySQL Community 版本是其开源商业模式的一部分[①]。开源商业模式中，并非一切都是免费的，部分需要付费，类似于部分付费游戏。换言之，在部分付费游戏中玩游戏是免费的，但是要想使用更多装备、技能就需要支付一定的费用。MySQL 与此类似，基本功能是免费的，而要使用更多高级功能就要支付费用，这种版本称为 MySQL 企业版。但是 MariaDB 没有这种区别[②]，用户不必支付额外费用以获得更多软件功能，所需功能可以随时下载并免费使用。具有一定能力的用户也可以下载源代码，通过源码分析了解各项功能实现的细节。当然，不包含软件之外的技术支持等服务。

MySQL 创始人 Monty 推出 MariaDB 开源数据库的原因如下：

- 希望 MySQL Code Base（以 MariaDB 的名义）继续保持开源；

① 当然，我们并不是对 Oracle 商业模式持否定态度，使用更好、更强大的功能当然需要付费。

② 现在，MariaDB 也推出了相应的企业版，使用时也需要支付一定费用。——译者注

- 希望为 MySQL 开发者搭建一个更好的平台，保证开发人员继续以开源方式开发 MySQL。

Monty Program AB 公司未来可能也会被其他公司收购，并因此取消开源。但是，从上面两点看，我认为：只要 Monty 还在，MariaDB 数据库就将继续保持开源，用户必需的功能也可以继续免费使用。此外，MariaDB 是在 MySQL 社区版基础上开发而成的，这会为 MariaDB 商业版的推出带来明显的法律风险。因为 Monty Program AB 公司并未从 Oracle 公司重新购回 MySQL。

本节对 MariaDB 进行了简单的介绍，但这对了解整个 MariaDB 还是远远不够的。由于本书并非主要阐述 MariaDB 与开源哲学，所以各位大致了解上面这些内容就可以了。

提示 “MariaDB”这个名称究竟从何而来？听着好像有点宗教意味，而作为 DBMS 的软件名称也有点不自然。要想弄清“MariaDB”源自何处，首先要了解 MySQL 是如何创建的，还要对 Monty 的家庭有一些了解，虽然这对我们并不重要。

很多人可能会对“MySQL”中的“My”一词产生误解，以为它是“I”（我）的所有格形式，即“我的”，于是将“MySQL”理解为“我的 SQL”。其实不然，“My”一词来自 Monty 的第一个女儿的名字。Monty 曾用二儿子名 Max 创建过名为 MaxDB 的 DBMS。介绍到这里，相信很多读者已经猜到了，MariaDB 源自 Monty 的第三个女儿 Maria。对于命名，Monty 似乎很有章法。Monty 生于 1962 年，到 2013 年已有 51 岁，应该不太可能推出第四个 DBMS 了。

1.2 MariaDB 与 MySQL

下面从计算机工程师（而非哲学）角度讲解 MariaDB。如前所述，MariaDB 始于 MySQL，通过比较二者异同，各位将进一步了解 MariaDB。

注意 本书所用的“MySQL”一词有时仅指 Oracle 的 MySQL，有时也指 MariaDB、PerconaServer（Percona 出品）、Oracle MySQL 的统称。因为对于篇幅及说明的简洁性而言，每次都使用“Oracle MySQL”或“MySQL 分支版本（Fork）”将会使表述变得冗长。这一规则也同样适用于 InnoDB 与 XtraDB 存储引擎。“InnoDB”有时仅指 Oracle MySQL 开发的 InnoDB 存储引擎，有时还包括 XtraDB。

我在叙述相关内容时会尽量使用合适的术语将之区分，但仍有一些内容无法通过用词的改变进行划分。各位阅读这些内容时，请根据上下文进行判断。

1.2.1 MariaDB、MySQL 和 PerconaServer

了解 MariaDB 之前，先要学习 MySQL 与 PerconaServer。相信各位已经掌握了大量 MySQL 服务器的相关知识。PerconaServer 是 Percona 公司（http://www.percona.com/）以 MySQL 服务器源代码为基础创建的 MySQL 服务器的另一个分支版本。

与 Oracle、Monty Program AB 不同，Percona 是一家以提供 MySQL 服务器咨询服务为主的公司。Percona 公司利用提供技术支持时积累的经验，对 MySQL 服务器存在的不足逐一进行修改调整，并在此基础上推出了 PerconaServer，大大提升了 MySQL 服务器的整体性能。

PerconaServer更注重数据库运营维护的便利与整体性能的提升。与MariaDB不同，每当MySQL发布新版本时，Percona会将自己扩展的功能插入MySQL服务器源代码，形成新的PerconaServer版本并发布。也就是说，Percona推出PerconaServer的周期以及版本号与MySQL几乎完全一致。

MySQL、MariaDB、PerconaServer三者之间的关系如图1-1所示。

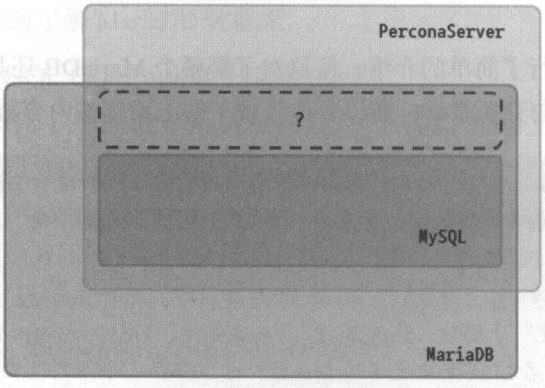


图1-1 MariaDB、PerconaServer之间的关系

从图1-1可以看到，MariaDB与PerconaServer都在MySQL服务器代码基础上推出新版本。现在，虽然MariaDB与MySQL源代码正逐渐向不同方向发展，但每当MySQL推出新版本时，PerconaServer几乎同时就会推出对应的新版本并添加新功能，对性能进行改善和提升。与PerconaServer类似，MariaDB也会在MySQL基础上添加一些扩展功能再推出。有意思的是，PerconaServer不会使用MariaDB中实现的新功能，但是MariaDB会把Percona中实现的功能（图1-1中的“?”部分）一起添加到新的发布版本，其中最具代表性的就是Percona开发的XtraDB存储引擎。

图1-2描述了MariaDB、MySQL、PerconaServer三者之间的版本关系。图中实线表示“基于”的关系，箭头所指的数据库是在箭尾数据库的基础上创建而成的；图中虚线表示“部分基于”的关系，即采用部分功能的源代码，并且这些代码一般都要经过修改完善。图中从PerconaServer的XtraDB到MariaDB的虚线表示MariaDB直接使用XtraDB存储引擎。

InnoDB是Oracle MySQL服务器默认包含的存储引擎，XtraDB是在InnoDB的源码基础上改良而成的新存储引擎。XtraDB与InnoDB的所有数据文件保持100%兼容。因此，实际应用中完全可以使用XtraDB取代InnoDB存储引擎。PerconaServer与MariaDB 5.5中，XtraDB取代了InnoDB，成为默认的存储引擎。到MariaDB 10.0.6[①]版本为止，尚未包含Percona的XtraDB存储引擎，但是当Percona的XtraDB存储引擎成为MariaDB 10.0必需功能后（MariaDB 10.0.7），它即作为默认存储引擎随MariaDB一起发布。因此，要先掌握并记住现在所用的

① 本书所讲内容均以MariaDB 10.0.6为基础，部分功能在MariaDB 10.0.7（开始与XtraDB一起发布）中测试。

MariaDB 中，内部使用的是 Oracle 的 InnoDB 存储引擎还是 Percona 的 XtraDB 引擎。

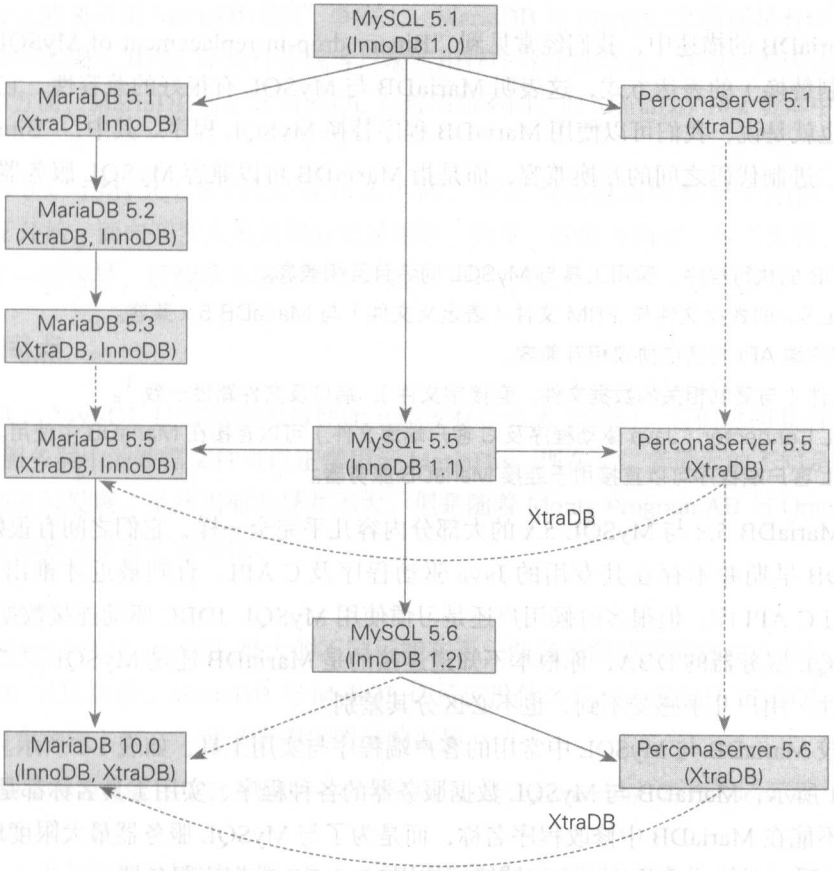


图 1-2 MariaDB、MySQL、PerconaServer 版本间的关系

以上简单介绍了 Oracle MySQL 与 Percona、Monty Program AB 推出的 MySQL 分支版本之间的关系。其实我真正想说的是，无论选用 MariaDB 还是 PerconaServer，若不了解 MySQL 的基本功能，都很难正常使用。MariaDB 与 PerconaServer 使用文档并不完善，现有文档仅对各自特有的功能进行了介绍，而对与 MySQL 服务器相同的功能并未编写或管理相应的使用文档。因此，使用 MariaDB 或 PerconaServer 时，不懂 MySQL 是绝对不行的。此外，从图 1-1 中还可以看到，MariaDB 内部使用了 XtraDB 存储引擎，所以有时还需要查看 Percona 的用户手册。

本节介绍了 3 个公司的数据库系统，它们拥有相同的基本代码，并分别添加了各自不同的功能。可能有些读者觉得有点难，但所幸的是，3 个公司都在尽最大的努力使自己的数据库与 Oracle MySQL 保持兼容。因此，学习使用 MariaDB 要比想的容易得多，请各位不要有太重的心理负担。

1.2.2 相同点

关于 MariaDB 的描述中，我们经常见到“Binary drop-in replacement of MySQL”（MySQL 的简易二进制替换）的表达方式，这表明 MariaDB 与 MySQL 有很好的兼容性，它们之间有很多共同点。也就是说，我们可以使用 MariaDB 程序替换 MySQL 程序。其中，“Binary”一词并不是指程序二进制代码之间的互换兼容，而是指 MariaDB 可以兼容 MySQL 服务器中使用的数据文件。

- MariaDB 的执行程序、实用工具与 MySQL 同名且互相兼容。
- MySQL 5.x 的数据文件与 .FRM 文件（表定义文件）与 MariaDB 5.x 兼容。
- 所有客户端 API 与通信协议相互兼容。
- 所有文件（与复制相关的数据文件、套接字文件）、端口及文件路径一致[①]。
- MySQL Connector（Java 驱动程序及 C 客户端库文件）可以直接在 MariaDB 中使用。
- MySQL 客户端程序可以直接用于连接 MariaDB 服务器。

其实，MariaDB 5.x 与 MySQL 5.x 的大部分内容几乎完全一样，它们之间有很好的兼容性，所以 MariaDB 早期并不存在其专用的 Java 驱动程序及 C API。直到最近才推出了 MariaDB JDBC 驱动与 C API 库，但很多时候用户还是习惯使用 MySQL JDBC 驱动连接数据库。如果不是管理 MySQL 服务器的 DBA，你根本不知道连接的是 MariaDB 还是 MySQL。二者之间拥有良好的兼容性，用户几乎感受不到，也不必区分其差别。

下面比较 MariaDB 与 MySQL 中常用的客户端程序与实用工具，如表 1-1 所示。

如表 1-1 所示，MariaDB 与 MySQL 数据服务器的各种程序、实用工具名称都是一样的。这并不是因为不能在 MariaDB 中修改程序名称，而是为了与 MySQL 服务器最大限度地保持兼容，使有 MySQL 服务器使用经验的用户能够轻松使用 MariaDB 数据库服务器。

表 1-1 程序与实用工具

	MariaDB	MySQL
数据库服务器程序	mysqld	mysqld
MyISAM 检查程序	myisamchk	myisamchk
客户端程序	mysql	mysql
服务器管理实用工具	mysqladmin	mysqladmin
数据导出程序	mysqldump	mysqldump
数据导入程序	mysqlimport	mysqlimport
内建的备份实用工具	mysqlhotcopy	mysqlhotcopy
Schema 更新实用工具	mysql_upgrade	mysql_upgrade
二进制日志分析程序	mysqlbinlog	mysqlbinlog
数据库结构分析实用工具	mysqlshow	mysqlshow
慢查询日志分析程序	mysqldumpslow	mysqldumpslow

① MariaDB 10.0 与 MySQL 5.6 中，用于实现复制的二进制日志格式略有差别。

MariaDB 与 MySQL 之间拥有如此高的兼容性，用户不禁会产生这样的疑问："既然两者高度兼容，为什么还要推出 MariaDB 呢?" 事实上，MariaDB 与 MySQL 之间还是有所不同的，然而这些不同并不体现在 JDBC 驱动程序或客户端实用程序等外显部分。在服务器优化器、存储引擎的设置内容方面，MariaDB 5.x 与 MySQL 有着明显不同。而 MariaDB 10.x 版本中，这些区别更加明显。

MariaDB 5.x 与 MySQL 5.x 几乎没什么不同，开发人员使用时甚至不用区分。但是从 MariaDB 10.x 开始，面向开发人员的部分在易用性、效率、性能方面有了很大改善，与 MySQL 相比逐渐有了一些差异，后续章节将逐步学习这些不同之处。

1.2.3 不同点

MariaDB 与 MySQL 目前在功能与性能上还没有什么太大不同，它们之间具有很好的兼容性，MySQL 服务器中的数据文件可以正常用于 MariaDB。现在，二者最大的不同是看谁能够引领各自开源项目的发展。虽然当前差异并不大，但是随着 Monty Program AB 与 Oracle 不同的取向发展，相信未来会出现更多变化。

1. 许可证

我认为 MariaDB 与 MySQL 最大的不同可能在于，两者开源政策的方向以及软件许可证政策。从 MySQL 社区版看，MariaDB 与 MySQL 许可证没什么差别；但是从 MySQL 的双许可政策（社区版与企业版）看，二者又表现出明显的差异。

- MariaDB 的发布遵循 GPL 许可协议，客户端程序及库（Connector/J 与 C Client API）遵循 LGPL 许可协议。
- MySQL 社区版与客户端程序及库（Connector/J 与 C Client API）都遵循 GPL 许可协议，MySQL 企业版遵循商业许可协议。

遵循 GPL 与 LGPL 协议的源码、库、可执行文件等都是开源共享的，但 LGPL 许可协议的限制要比 GPL 许可协议宽松一些。使用 MySQL 服务器的源代码时要遵循 GPL 许可协议，将 MariaDB 或 MySQL 源代码打包到商业软件并发布时，会受到很大影响。进行软件打包时，MySQL 客户端库文件是必需的部分。由于 MariaDB 的客户端库文件遵循 LGPL 许可协议，所以使用时更加自由、方便。

此外，MariaDB 的发布遵循 GNU GPLv2 许可协议，且无单独的商业版本。由于 MariaDB 的源代码建立在 Oracle MySQL 的源代码基础之上，这就限制了 Monty Program AB 推出 MariaDB 商业版。但对于使用 MariaDB 的用户而言，这恰恰是 MariaDB 的优点。

2. 存储引擎

从功能上说，MariaDB 与 MySQL 最大的不同可能是存储引擎。从编写 SQL 语句角度看，它们没有什么不同，但是 MariaDB 与 MySQL 存储引擎本身拥有不同用途。

表 1-2 MariaDB 与 MySQL 存储引擎比较

用途	MariaDB	MySQL
默认内存存储引擎	MySQL 将代码库中包含的 MEMORY 存储引擎用作默认内存存储引擎。MariaDB 中的 MEMORY 存储引擎经过了改善，可以支持 VARCHAR 可变数据类型，MariaDB 用户手册中提到了这一点。也就是说，若 MariaDB 的内存表或内部临时表中存在 VARCHAR(2000) 列，MariaDB 就会消耗与实际存储值同等长度的内存。但确认后发现，MariaDB 10.0 版本中并未实现该功能	MySQL 5.6 的内存存储引擎功能并没有太大变化，几乎与 MariaDB 的内存存储引擎一样
基于磁盘的内部临时表存储引擎	MariaDB 默认采用 Aria 存储引擎作为内部临时表存储引擎。与 InnoDB 类似，Aria 存储引擎可以使用内存缓存处理索引、记录数据等，所以比 MyISAM 的处理速度要快得多。用户可以通过 aria_used_for_temp_ tables 系统选项为内部临时表指定存储引擎，以决定使用 MyISAM 还是 Aria 存储引擎（其实，不是通过 aria_used_for_temp_tables 系统选项调整的，而是仅通过 MariaDB 选项进行调整。更多内容将在 Aria 存储引擎部分进行详细讲解）	MySQL 服务器需要使用内部临时表时，就会使用 MyISAM 存储引擎创建并处理表
支持事务处理的存储引擎	MariaDB 使用 XtraDB 而非 InnoDB 作为支持事务的存储引擎。关于 XtraDB 会在后面存储引擎部分进行详细介绍。从 MariaDB 10.0.7 开始，使用 Oracle 的 InnoDB 而非 XtraDB 作为默认存储引擎。XtraDB 存储引擎的使用方法会在后面进行详细说明	MySQL 中常用的 InnoDB 为默认存储引擎，它也是一款支持事务处理的存储引擎
支持 NoSQL 的存储引擎	从 MariaDB 10.0 开始，在 MariaDB 外部连接正在运行的 Cassandra 后，通过 MariaDB 服务器可以访问 Cassandra 的数据	从 MySQL 5.6 开始提供 Memcached 插件，它不是存储引擎，但通过 Memcached API 可以快速访问 InnoDB 的数据

像这样，由于 MariaDB 与 MySQL 内部使用不同存储引擎，所以两者的性能调优选项彼此不同。每个存储引擎都有各自的特点，它们也有不同的优缺点。

3. 功能

MariaDB 5.5 与 MySQL 5.5 在功能上没有什么不同。MySQL 服务器中新增的功能用不了多久就会出现在 MariaDB 中。MariaDB 从 MySQL 分离出来的时间并不长，它们添加功能的先后顺序几乎是一样的。MariaDB 与 MySQL 各自拥有的独立功能没有多少，如表 1-3 所示。大多数时候，它们的区别只是同一功能被添加到哪个版本的 MariaDB 与 MySQL 而已。当然，也有些功能是 MySQL 支持而 MariaDB 社区版不使用的。

表 1-3 MariaDB 与 MySQL 的功能比较

	MariaDB	MySQL
线程池	从 MariaDB 5.1 开始支持	MySQL 社区版不支持，仅 MySQL 5.5 企业版支持
缓冲池预加载	MariaDB 5.5 中的 XtraDB 可以转储缓冲池中的内容，重启 MariaDB 后，缓冲池中转储的内容会再次加载到缓冲池	从 MySQL 5.6 开始支持 InnoDB 缓冲池的转储与加载功能
支持 SSD	MariaDB 5.5 的默认存储引擎 XtraDB 支持 Block flush 算法，从而支持基于 SSD 的磁盘 I/O	-
子查询优化	从 MariaDB 5.5 开始支持 “IN（subquery）” 与 “NOT IN（subquery）” 优化，也支持将 FROM 语句中的子查询（DERIVED）与外部查询合并（Query rewriting）优化功能	从 MySQL 5.6 开始支持 “IN（subquery）”、“NOT IN（subquery）”，以及 FROM 语句子查询优化功能

（续）

	MariaDB	MySQL
连接优化	支持基于增量连接缓冲的 Nested Loop 连接与 BKA 连接，默认支持的方式为散列连接	支持基于增量连接缓冲的 Nested Loop 连接与 BKA 连接。MySQL 5.6 不支持散列连接
多源复制	在传统复制模式下，一个从服务器无法对应两个以上的主服务器。从 MariaDB 10.0 开始支持多源复制，即一个从服务器可以有多个主服务器	从 MySQL 5.7 版本开始支持多源复制
多线程复制	之前的复制，从服务器只能利用 1 个线程处理主服务器的复制查询，所以经常发生从服务器的复制延迟现象。从 MariaDB 10.0 开始，复制线程个数设置为 2 个以上，各线程执行域名 ID 为单位的并行复制	从 MySQL 5.6 开始支持多线程从服务器，它不以记录为基础，而采用数据库单位的多线程方式。若 MySQL 服务器中仅存在 1 个数据库，则以之前的单线程方式进行复制
基于角色的权限管理	从 MariaDB 10.0 开始采用类似于 Oracle RDBMS 的方式为特定用户组创建角色，将各个角色赋予用户以进行权限管理	–
微秒级数据类型	从 MariaDB 5.3 开始，用户可以将以毫秒与微秒为单位的时间存储到 TIME 或 DATETIME 类型，并能从中读取	从 MySQL 5.6 开始拥有与 MariaDB 相同的功能
半同步复制	–	从 MySQL 5.5 开始以插件形式添加该功能
虚拟列	从 MariaDB 5.2 开始支持虚拟列，使用虚拟列功能可以将 1 个以上的列值事先存储到另外的列，进行查询处理时再加工并显示	–
动态列	像 NoSQL 一样，各记录拥有的列不固定时，常常会使用动态列。动态列由 1 个物理列形成，在该物理列中可以创建多个逻辑列，并进行值的保存或读取	–
PAM 认证	从 MariaDB 5.2 开始，用户可以使用插入式认证模块	MySQL 社区版不支持该功能，仅在 MySQL 5.5 以上的企业版中支持

4. 优化器

MariaDB 与 MySQL 服务器的优化级别是不可见但又不可忽视的部分。优化部分在 MariaDB 5.3 中发生了很大变化，它的优化选项被应用于 MySQL 5.6、5.7 版本。由于 MySQL 5.7 尚未发布[①]，下面仅比较 MySQL 5.6 与之前几个版本的主要优化选项与功能。

表 1-4 比较各版本支持的功能

类别	功能	MariaDB 5.3/5.5	MariaDB 10.0	MySQL 5.5	MySQL 5.6
读磁盘	Multi-Range Read（MRR）	YES	YES	–	YES
	index_merge 执行计划的 Sort_intersection 算法	YES	YES	–	–
	基于成本的 range 与 index_merge 执行计划选择	YES	YES	–	–

① 原书出版于 2014 年 4 月。——编者注

（续）

类别	功能	MariaDB 5.3/5.5	MariaDB 10.0	MySQL 5.5	MySQL 5.6
读磁盘	ORDER BY...LIMIT〈small_limit〉	–	YES	–	YES
	辅助索引扩展（Extended secondary index optimization）	YES(5.5)	YES	–	YES
连接	Batched key access（BKA）	YES	YES	–	YES
	Block hash join	YES	YES	–	–
子查询	IN 查询优化（In-to-exists optimization）	YES	YES	YES	YES
	半连接（Semi-join）	YES	YES	–	YES
	具体化（Materialization）	YES	YES	–	YES
	基于成本的具体化与 In-to-Exist 优化（Cost based choice of materialization vs.. in-to-exists）	YES	YES	–	YES
	子查询缓存（Subquery cache）	YES	YES	–	–
临时表 & 视图	派生表与视图延迟具体化（Delayed materialization of derived tables/ materialized views）	YES	YES	–	YES
	派生表的执行计划性能提升（Instant EXPLAIN for derived tables）	YES	YES	–	YES
	派生表索引优化（Derived Tabled with Keys optimization）	YES	YES	–	YES
其他	LIMIT ROWS EXAMINED rows_limit	YES(5.5)	YES	–	–
	DELETE、INSERT、REPLACE、UPDATE 执行计划	–	实现中	–	YES
	以 JSON 格式输出执行计划	–	–	–	–YES

前面讲解 MariaDB 与 MySQL 优化器的不同时，并未涉及代码内部，仅对易觉察的部分进行了简单的介绍。从介绍可知，MariaDB 5.5 中的优化选项被应用于 MySQL 5.6，增强了对于子查询与派生表创建查询的优化。以上内容将在后面通过示例逐一讲解。

1.2.4 各版本间的兼容性

MariaDB 与 MySQL 的大部分功能是类似的，相互具有良好的兼容性，相同的数据文件、日志文件几乎都可以在各版本之间正常使用。各版本兼容情况如表 1-5 所示。事实上，MariaDB 5.1、5.2、5.3 是基于 MySQL 5.1 的源代码创建的，而 MariaDB 5.5 则基于 MySQL 5.5、MariaDB 5.3 的代码创建而成。MariaDB 10.0 基于 MariaDB 5.5 的代码，而非 MySQL 5.6。

表 1-5 MariaDB 与 MySQL 版本兼容表

MariaDB	MySQL
5.1、5.2、5.3	5.1
5.5、10.0	5.5

如表 1-5 所示，MySQL 5.1 与 MariaDB 5.1~5.3 兼容，MySQL 5.5 与 MariaDB 5.5~10.0 兼容。此处的“兼容”是指，MariaDB 的各版本是在相应版本的 MySQL 源代码基础上创建而成的，还表示大部分数据文件、客户端与服务器端的通信协议、相关实用工具均可通用。

但是，MariaDB 与 MySQL 也不完全一样。虽然它们大部分内容是相互兼容的，但是暴露给用户的部分仍存在差别，有些部分的操作也略有不同。下面看看各版本需要关注的部分。

1. MariaDB 5.1 与 MySQL 5.1

- MariaDB 的安装程序（rpm 或 msi）名称由 MySQL 变为 MariaDB；
- MariaDB 的 mysqld 程序也可以从设置文件中识别“[mariadb]”节区；
- 处理 CHECKSUM TABLE 命令时，MariaDB 不会忽略 NULL 列值，所以可能得到与 MySQL 不同的校验和；
- MariaDB 的慢查询日志会将各查询的简单执行计划（关于执行全表扫描还是使用临时表等基本信息）一并显示；

MariaDB 在内部使用 Aria 存储引擎，而不是 MyISAM 存储引擎，Aria 存储引擎占用的内存更多。

2. MariaDB 5.2 与 MySQL 5.1（包含 MariaDB 5.1 与 MySQL 5.1 的不同）

将 IGNORE_BAD_TABLE_OPTIONS 选项添加到 SQL_MODE。若没有该项，存在存储引擎无法识别的表选项时，将产生错误。

3. MariaDB 5.3 与 MySQL 5.1（包含 MariaDB 5.2 与 MySQL 5.1 的不同）

- 仅在 MariaDB 中发生的错误代码从 1900 开始（防止与 MySQL 错误号发生冲突）；
- 借助 DATETIME 与 TIME 数据类型可以将时间精确到微秒级；
- UNIX_TIMESTAMP() 函数返回的时间戳可以精确到小数点后 6 位；
- SHOW PROCESSLIST 命令返回的结果中添加了 Progress 表项，用于显示查询处理进展情况；
- IGNORE 选项在安全情形下被忽略，除此之外（发生 FATAL 错误时）会发生错误；
- 添加了大量查询优化器功能（optimizer_switch 选项），增加了优化器优化方式调优功能。

4. MariaDB 5.5 与 MySQL 5.5

- INSERT IGNORE 中发生重复键错误时，会出现警告信息。

以上简单介绍了 MariaDB 与 MySQL 各版本兼容与非兼容的内容。其实它们之间的非兼容部分非常少，从我多年的使用经验看，这些微小差异不会对主要功能的使用产生太大影响。

1.2.5 性能比较

各位可能非常关心 MariaDB 与 MySQL 的性能比较，选择其他解决方案时也会用类似的标准，但几乎没有人能够轻松说出 MariaDB 与 MySQL 哪个更快。要回答这个问题需要考虑很多方面（如下所示），所以本书将不会对此进行讨论。

- 测试方案在 MariaDB 与 MySQL 中是否都能以最佳方式运行？最佳标准是如何应用的？
- MariaDB 与 MySQL 服务器是否都调整为最佳状态？
- 是否充分考虑了 MariaDB 和 MySQL 的执行性能最优部分？
- 在相当长的时间内表现出稳定性能的性能测试结果是收集的？

在网上输入“MariaDB MySQL Performance Comparision”关键字搜索时，会看到许多测试结果。如果时间允许，可以进入这些网站查看 MariaDB 与 MySQL 的性能比较结果。图 1-3 和图 1-4 这两个曲线图分别来自 MariaDB（https://mariadb.com/blog/sysbench-oltp-mysql-56-vs-mariadb-100）与 Planet MySQL（http://planet.mysql.com/entry/?id=36221）的基础测试图。

各位可以从上图了解我的意图。从图可以看出，测试结果会根据测试人以及所用测试方案的不同而不同。即便如此，我们也不能怀疑测试人员进行这种测试的能力。可以肯定的是，它们都是由 MySQL 顶级专家所做的测试。其实，如果让我进行测试，我也很难比专家更精确、严密。

如何从上述测试结果判断 MariaDB 和 MySQL 哪个的性能更好呢？我认为，可以亲自用自己公司的服务模型测试 MariaDB 与 MySQL 并比较测试结果。但这就要求所选数据库的服务器选项与测试方案都是最合适的。事实上，从如此类似的 RDBMS 中选择一个是相当困难的。从图 1-3 与图 1-4 的测试结果中可以明显看到，有一项指标至关重要。两个测试结果表明 MariaDB 与 MySQL 性能差别不大，不足以对选择 MariaDB 还是 MySQL 产生决定性的影响。那么来想一想，我们提供的所有服务中，是否有服务带有“每秒事务处理量不能为 3000，必须达到 3500”这种强约束条件呢？其实，几乎不存在这种情形。对于廉价设备中运行的 MySQL 服务器而言，其性能通常由硬件性能决定，所以不必太纠结于这些小的性能差异，也没有必要比较 MariaDB 与 MySQL 的性能。

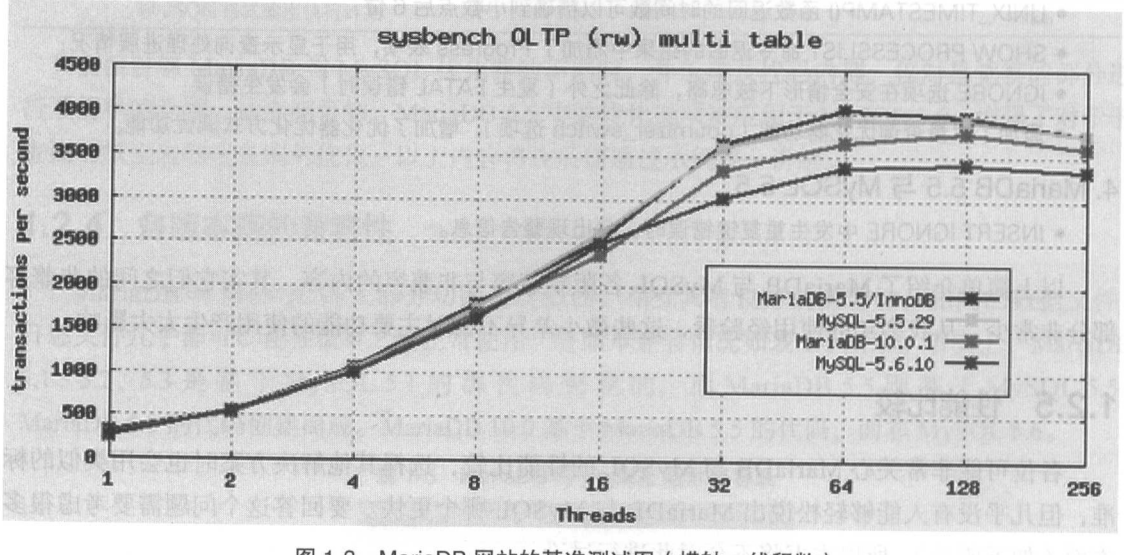


图 1-3 MariaDB 网站的基准测试图（横轴 = 线程数）

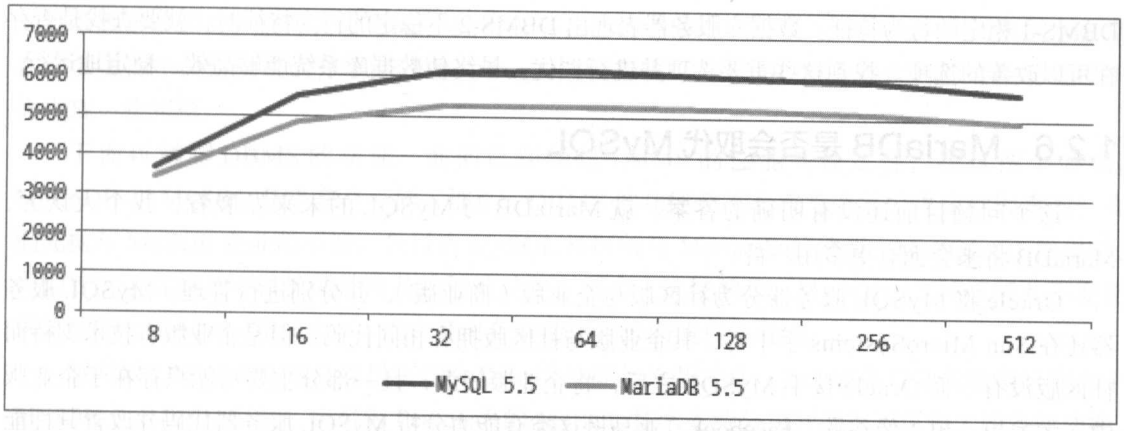


图 1-4 Planet MySQL 网站的基准测试图（横轴 = 线程数）

如果有 DBMS 的实际运营经验或长期使用经验，那么你必定对一个项目有切肤之痛，那就是“稳定性”。上面两个测试只比较了两个数据库系统在单位时间内的业务最大处理量（事务数或查询数），但对于向外提供服务的数据库服务器而言，保持其性能的稳定性是最重要的。因为我们往往需要其长时间运行——短则几个月，长则几年——且在运行期间内保持稳定性能。

图 1-5 不是实际的测试结果，而是我为了说明 DBMS 的稳定性而绘制的性能曲线图。图中曲线表示 DBMS-1 与 DBMS-2 两个 DBMS 服务器每秒的事务处理量，在 11:00~11:10 这个时间段内，两个 DBMS 的总事务处理量（Throughput，吞吐量）是一样的。但是从某个时间点看，有时 DBMS-2 要比 DBMS-1 性能高，有时又比 DBMS-1 的性能低。此时可以做出如下判断：DBMS-1 每秒能够持续处理 100 个左右的事务，而 DBMS-2 每秒只能处理 60 个（曲线谷值）左右的事务。因为在一般的服务环境中，一定级别的事务请求会持续到来。

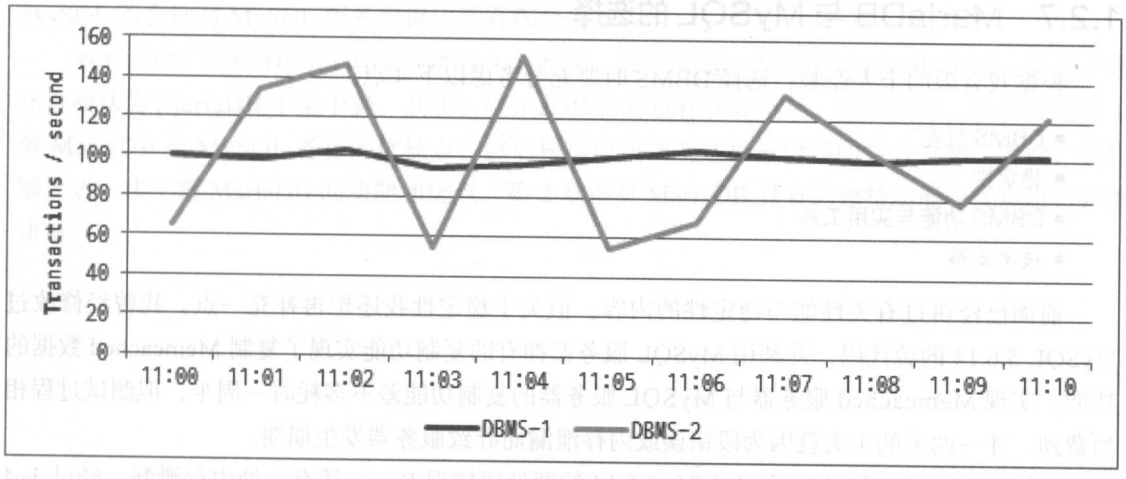


图 1-5 DBMS 处理稳定性比较

图 1-5 中的 DBMS-1 与 DBMS-2 并非对应于 MariaDB 与 MySQL 服务器。根据各种服务特点，MariaDB 与 MySQL 服务器有时表现出 DBMS-2 上下波动的行为特征，有时又表现出

DBMS-1 稳定的行为特征。数据库服务器表现出 DBMS-2 不稳定的行为特征时，就要查找是否存在可以改善的选项。找到这些重要选项并进行调优，最终使数据库系统能够高效、稳定地运行。

1.2.6 MariaDB 是否会取代 MySQL

这个问题目前还没有明确的答案。就 MariaDB 与 MySQL 的未来发展看，我个人认为，MariaDB 将来会拥有更多用户群。

Oracle 将 MySQL 服务器分为社区版与企业版（商业版），并分别进行管理。MySQL 服务器还在 Sun MicroSystems 手中时，其企业版与社区版拥有相同代码，但是企业版有技术支持而社区版没有。而 Oracle 接手 MySQL 之后，将企业版闭源，且一部分主要功能只存在于企业版中。大家想一想，像谷歌、Facebook、亚马逊这类有能力分析 MySQL 服务器代码并改善其性能的公司，以后会怎么做呢？在 MariaDB 与 MySQL 中，他们会选择哪一个？上面提及的三家公司会对整个 MySQL 生态界产生极大影响。

MariaDB 具有的最大优点是，它拥有比 MySQL 更丰富的功能，更新快，且开源，并与 MySQL 保持良好的二进制兼容。一直使用 MySQL 服务器的公司可以很容易地从 MySQL 迁移到 MariaDB。向 MariaDB 迁移的过程中，甚至不需要修改应用程序的任何代码，MySQL 拥有的功能同样可以在 MariaDB 中正常使用，因此，用户几乎不用担心从 MySQL 迁移到 MariaDB 会出现什么问题。

谷歌声称将支持 MariaDB 开源，这表明 MariaDB 不会轻易消失。但还要看 Oracle 以后的动向，看它是将主要精力用于 MySQL 企业版（闭源）的功能添加与性能提升，还是用于使社区版更具活力。

1.2.7 MariaDB 与 MySQL 的选择

根据我有限的个人经验，选择 DBMS 时要充分考虑以下 4 点：

- DBMS 性能
- 稳定性
- DBMS 功能与实用工具
- 技术支持

前面已经讲过有关性能与稳定性的内容，但关于稳定性我还想再补充一点。我曾经修改过 MySQL 5.6.14 的源代码，并利用 MySQL 服务器拥有的复制功能实现了复制 Memcached 数据的功能。实现 Memcached 服务器与 MySQL 服务器的复制功能差不多耗时一周半，但测试过程相当费劲，才一两天的工夫就因为段错误或内存泄漏而导致服务器发生崩溃。

实际测试期间，我发现了 MySQL 5.6.14 的两处段错误 Bug，还有 3 处内存泄漏。经过 3~4 周的调试并修改相关代码后，服务器才得以稳定运行。发现内存泄漏 Bug 后并提交给 Oracle 后，Oracle 要花费 3 周以分析出现 Bug 的原因，而针对这些 Bug 的补丁也不知什么时候才能添

加到新版本。因此，我们发现 Bug 后经常需要自己动手修改。不仅 5.6.14 版本，其他版本也存在许多类似于段错误或内存泄漏的重大 Bug，这使得测试 MySQL 服务器代码稳定性的过程中会出现一些问题。

下面再聊聊 DBMS 的功能。前面已经谈过 DBMS 的性能与稳定性，在这两个方面，MariaDB 与 MySQL 几乎完全一样。那么，DBMS 的功能会不会成为选择的决定性因素呢？我们先抛开 MySQL 企业版不谈，只讨论 MySQL 社区版与 MariaDB 的功能。MySQL 5.5 推出时，人们曾大力宣扬其多核扩展性以及线程池功能。但令人遗憾的是，MySQL 5.5 社区版并不能使用该功能。而 MariaDB 5.5 中，用户可以免费使用类似的线程池功能。

有时候，MariaDB 还会提供更好的 DBMS 功能与诊断工具。企业级备份功能原本只存在于 MySQL 企业版中，但是一家名为 Percona[①] 的 MySQL 技术支持公司开发出了更强大的备份功能，并发布给用户，供大家免费使用。用户有时想知道 MySQL 服务器的内存使用情况，或想知道哪些线程占用了更多内存。这些小功能虽然琐碎，但却是必需的。在这些方面，MariaDB 提供了比 MySQL 更多的功能以满足用户需求。对于这些小而必需的功能，Oracle 或 Monty Program AB 这样的 RDBMS 供应商开发较少；而使用 MySQL 或 MariaDB 的大企业则开发较多，比如 Facebook、谷歌等。MariaDB 没有商业版本，所以这些增强功能谁都可以随意使用，不必受许可证限制。这最终会使 MariaDB 具有更丰富的功能。

最后谈谈有关技术支持的内容。要获得 MySQL 技术支持，用户需要与 Oracle 签订合同。同样，用户要想获得 MariaDB 技术支持，也需要与 SkySQL[②] 签订合同。Oracle 只针对 MySQL 服务器提供技术支持，SkySQL 除了提供对 MariaDB 的技术支持外，还对 MySQL 提供技术支持。对于用户对 MySQL 服务器的疑问，SkySQL 会分析问题所在并查明原因。若有可用的替代方案，则告知用户；反之，则他们只会向管理源代码的 Oracle 提交 Bug 报告。也就是说，SkySQL 不会针对 MySQL 服务器提供所有部分的支持。

本节讨论了选择 DBMS 时要考虑的 4 个方面。虽然这是一本关于 MariaDB 的书，但并不意味着我认为 MariaDB 十全十美。其实，MariaDB 自身也存在着不足。重要的是，各位要充分了解 MariaDB 与 MySQL 各自的优缺点，然后根据应用环境选择最恰当的结合方案。为此，我们需要进一步了解 MariaDB 的功能和优点。第 2 章将对 MariaDB 具有的独特功能与优点进行详细讲解。

---

① Percona 公司开发并推出了 PerconaServer 服务器，它是 MySQL 的一个分支版本。本书主要讲解 MariaDB，故不会详细讲解 PerconaServer 服务器。后面讲解 XtraDB 存储引擎时，会对其进行简介。

② Monty Program AB 的技术支持公司。

第2章

安装

2.1 下载
2.2 安装
2.3 更新

2.1 下载

首先打开浏览器，输入如下地址，进入 MariaDB 官方网站的下载页面，下载 MariaDB 安装程序。

https://downloads.mariadb.org/

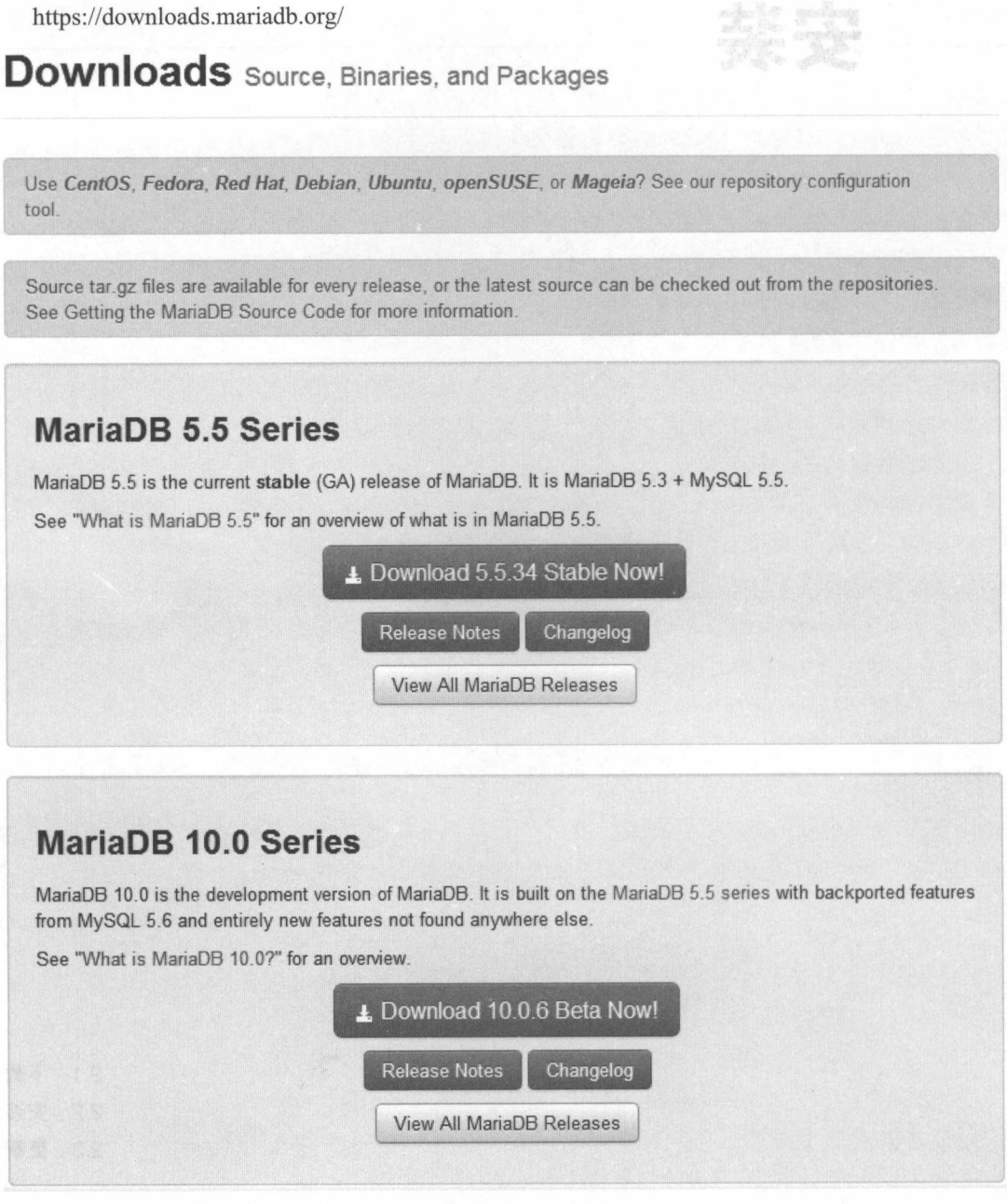


图 2-1 MariaDB 下载页面

如图 2-1 所示，下载页面中存在多个版本的 MariaDB 下载链接。目前 MariaDB 已经更新到 10.0 版本，当然，如果你喜欢使用 MariaDB 5.5 系列版本，可以单击下载 MariaDB 5.5 稳定版，进入图 2-2 所示页面，页面列出了 MariaDB 5.5 版本在各平台上的包目录。若想下载以前版本的 MariaDB 服务器，请单击页面右上角的“View all release”按钮，进入所有版本页面，列出 MariaDB 5.5 之前的所有版本。其中，Tar 文件（mariadb-5.5.34.tar.gz）是 MariaDB 源文件，其后没有注明所用平台，下载后可以直接进行编译。

Tar 格式的下载文件名称中，各平台的简称含义如表 2-1 所示。

表 2-1 平台简称含义

winx32	32 位 Windows 操作系统
winx64	64 位 Windows 操作系统
linux-i686	32 位 Linux 系列操作系统（含 CentOS、RedHat、Fedora）
linux-x86_64	64 位 Linux 系列操作系统（含 CentOS、RedHat、Fedora）

MariaDB 5.5.34 Stable 2013-11-21

View all releases

Release Notes | Changelog

Affordable,enterprise class product support, professional services, and training for your MariaDB database is available from the MariaDB Foundation's release sponsor, SkySQL Ab. To learn more about them and their services for MariaDB, visit their website, or email SkySQL at sales@skysql.com

File Name	Package Type	OS / CPU	Size	Meta
mariadb-5.5.34.tar.gz	source tar.gz file	Source	45.7 MB	MD5 Instructions
mariadb-5.5.34-winx64.zip	ZIP file	Windows 64-bit	141.6 MB	MD5 Instructions
mariadb-5.5.34-winx64.msi	MSI Package	Windows 64-bit	56.7 MB	MD5 Instructions
mariadb-5.5.34-win32.msi	MSI Package	Windows 32-bit	54.9 MB	MD5 Instructions
mariadb-5.5.34-win32.zip	ZIP file	Windows 32-bit	142.2 MB	MD5 Instructions

Want to learn more about MariaDB? Check out our whitepapers.

Operating System

- DEB Package
- Generic Linux
- RPM Package
- Source Code
- Windows

图 2-2 各平台下的 MariaDB 包

下载页面最底部专门为 Debian 或 Ubuntu 提供了下载包，还为其他 Linux 系列提供安装包下载链接。单击“Red Hat, Fedora, and CentOS Packages”链接，根据不同国家或地区链接到相应 FTP 站点，再从相关网站下载合适版本的 RPM 包即可。RPM 包面向不同平台，x86 为 32 位操作系统，amd64 为 64 位操作系统。其实，可以将 amd64 理解为 x86_64，它们是相同的表示方式。

RPM 包是根据不同模块分别进行打包的，每个包包含如下程序与库文件。如果各位选择 TAR 压缩文件下载 MariaDB（*.tar.gz），则压缩文件中已经包含了以下所有程序与库文件（Galera Cluster 相关程序除外），所以只要下载 tar.gz 一个压缩文件即可。

表 2-2 RPM 包中的文件

MariaDB- 版本号 - 操作系统 -CPU 类型 -client.rpm	启动 MariaDB 客户端所需要的程序与文件
MariaDB- 版本号 - 操作系统 -CPU 类型 -common.rpm	启动 MariaDB 服务器或客户端所需的设置文件（my.cnf），或定义字符集的 XML 文件
MariaDB- 版本号 - 操作系统 -CPU 类型 -compat.rpm	用于实现兼容的共享库，使得与以前版本的 MariaDB 库一起编译的外部客户端可以启动运行
MariaDB- 版本号 - 操作系统 -CPU 类型 -devel.rpm	使用 MariaDB 编写 C/C++ 程序时所需的头文件与库文件
MariaDB- 版本号 - 操作系统 -CPU 类型 -server.rpm	MariaDB 服务器
MariaDB- 版本号 - 操作系统 -CPU 类型 -shared.rpm	运行动态链接 MariaDB 库文件的外部程序时所需的库文件
MariaDB- 版本号 - 操作系统 -CPU 类型 -test.rpm	用于测试 MariaDB 服务器的程序
MariaDB-Galera- 版本号 - 操作系统 -CPU 类型 -server.rpm	MariaDB+Galera Cluster 安装程序
galera- 版本号 - 操作系统 -CPU 类型 .rpm	MariaDB+Galera Cluster 的 Galera 库

一般而言，只要安装好 common 与 server RPM，MariaDB 服务器即可正常启动。要启动 mysql 客户端，则要安装 common 与 client RPM。使用 C/C++ 编程语言编写访问 MariaDB 服务器的外部程序时，为了使用共享库，编译程序时要安装 shared 与 compat RPM。MariaDB 服务器与客户端都是静态编译后发布的，所以运行 MariaDB 服务器或客户端时，不需要 shared 与 compat RPM 共享库。

图 2-1 所示页面中，向上滚动页面可以看到下载 MariaDB Java JDBC 驱动程序与 C/C++ 客户端库的链接，需要时单击下载即可。Java JDBC 驱动程序和 C/C++ 客户端库与 MariaDB 版本相互独立，这意味着，连接 MariaDB 5.5 服务器时不必非得使用 5.5 版本的客户端。使用 Java 驱动程序或 C/C++ 客户端库时，下载最新版本使用即可，不会出现版本差异的问题。

2.2 安装

MariaDB 支持多种平台，由于篇幅所限，我们无法逐一学习各操作系统下的安装方法。我们通常会在搭载 Windows 操作系统的个人 PC 上安装 MariaDB，以进行测试，而实际用于向外提供服务的 MariaDB 服务器主要安装在 Red hat 或 CentOS 等 Linux 操作系统中。因此，此处只

讲解将 MariaDB 安装到 Windows 与 Linux 操作系统（以 CentOS 为例）中的方法。

2.2.1 在 Windows 操作系统下安装 MariaDB

Windows 操作系统下的 MariaDB 安装方法有两种，一种是使用 MSI 安装程序，另一种是使用 ZIP 压缩包。MSI 安装程序会将 MariaDB 服务器注册为 Windows 服务，包含 MariaDB 服务器安装向导等。使用 ZIP 压缩包安装 MariaDB 时，不需要注册 Windows 服务，但需要手动编写 MariaDB 设置文件（my.ini）。

注意 包含 XtraDB 的 PerconaServer 并无针对 Windows 的版本。即使在 Windows 中安装了 MariaDB，也无法使用 XtraDB 存储引擎。若想测试或学习 XtraDB 的性能及功能，只要将 MariaDB 安装到 Linux 系统即可。由于 Percona 公司尚无开发 Windows 版本的计划，所以用户还无法在 Windows 系统中使用 XtraDB 或 PerconaServer。

1. 使用安装程序（MSI）进行安装

本书介绍的所有内容均基于 MariaDB 10.0，所以先要下载最新的 MariaDB 10.0.x Windows 安装程序。虽然安装方法不怎么难，但与在 Linux 系统下安装不同，用户需要在安装过程中选择几个环境设置项。

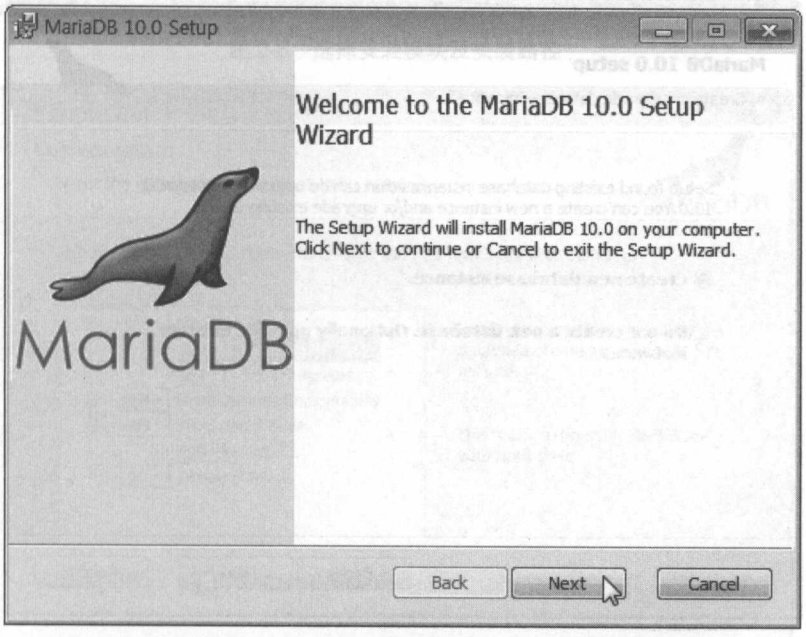


图 2-3 开始安装

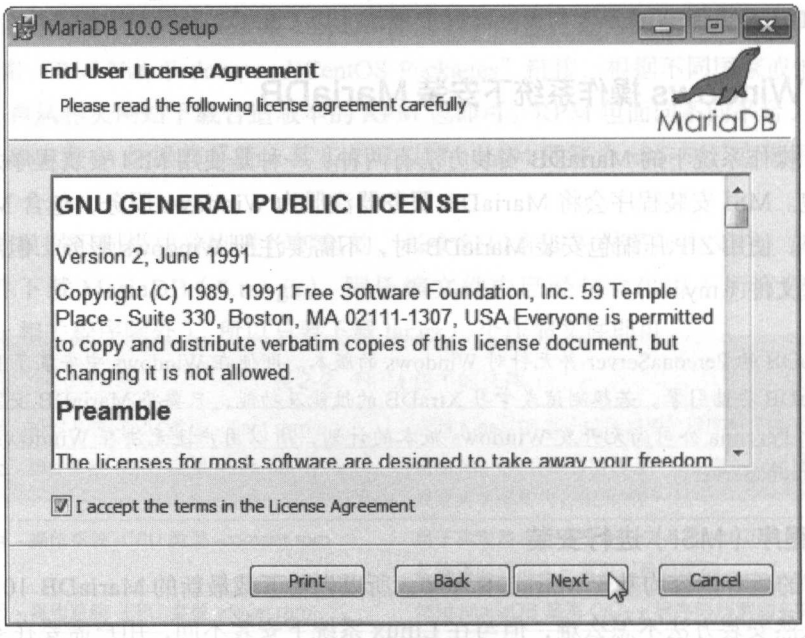


图 2-4 许可证协议

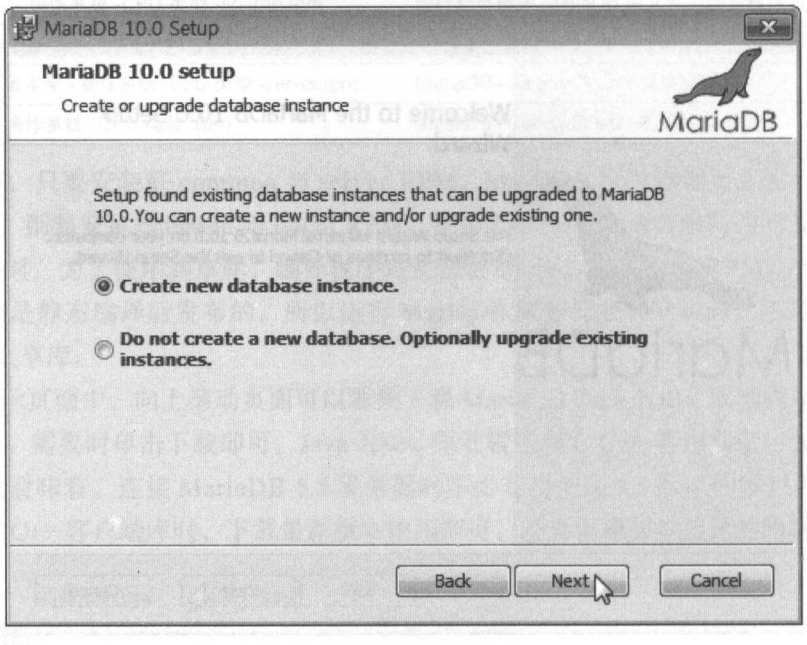


图 2-5 选择安装或更新

图 2-6 所示的安装过程中，用户可以自由选择安装 MariaDB 服务器程序的相关功能，并指定安装路径。虽然选择 MariaDB 程序的安装路径不怎么重要，但是如果想用大量数据测试 MariaDB 或以服务形式运行 MariaDB 时，应当在图 2-7 所示的数据目录选择界面中选择备有数

据目录的磁盘卷。当然，安装完成后，安装过程中所选的数据目录及缓冲池选项可以在MariaDB 服务器的设置文件（my.cnf）中再次进行修改。

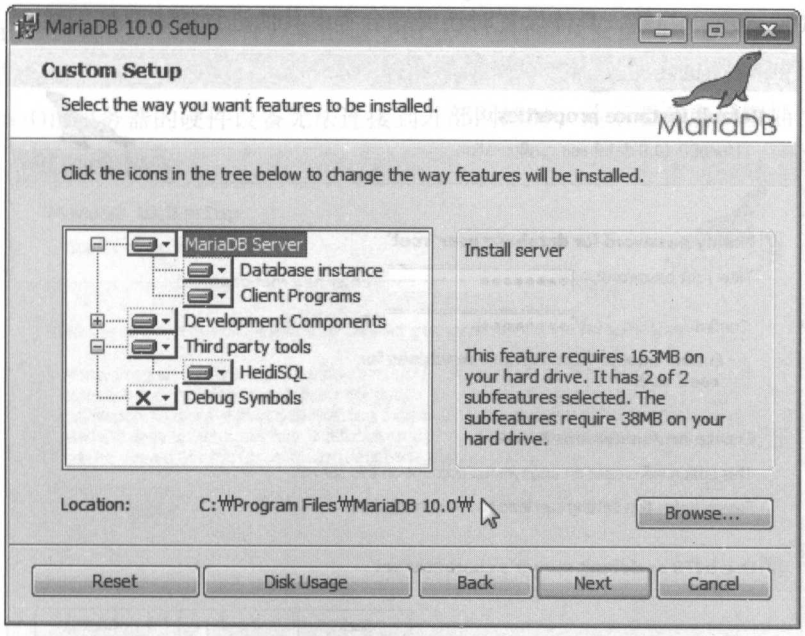


图 2-6 选择安装模块及安装路径

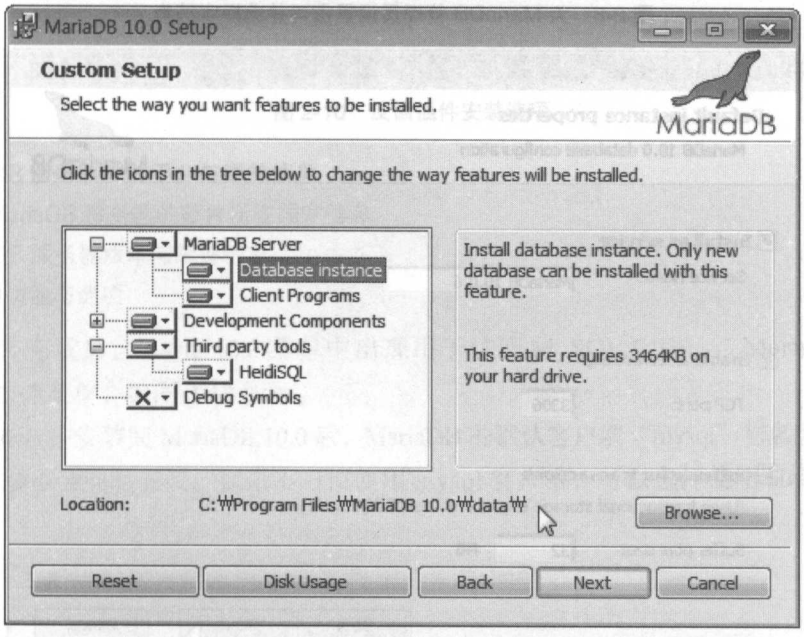


图 2-7 选择数据目录

图 2-9 所示对话框中，用户可以设置 MariaDB 服务器的端口以及 InnoDB/XtraDB 缓冲池大

小。若 PC 中安装的 MariaDB 仅供测试使用，则将缓冲池大小设置为 32~64 MB 即可。但若实际应用环境中安装了 MariaDB 服务器，并且用于向外提供服务，则此时缓冲池的大小应设置为整个内存的 50%~80%。当然，具体设置时还要综合考虑其他进程的内存使用量。

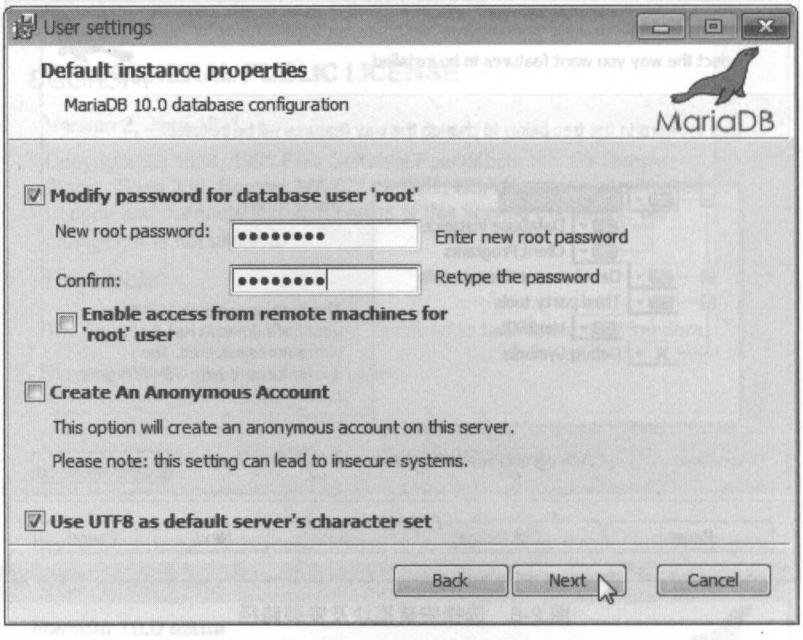


图 2-8 为 MariaDB 管理员设置密码并选择字符集

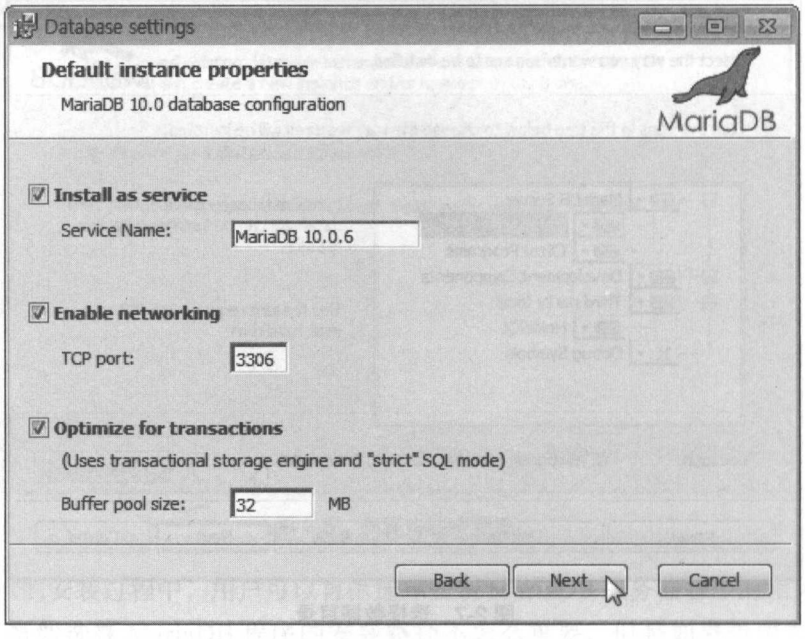


图 2-9 设置 MariaDB 服务器端口与 InnoDB/XtraDB 缓冲池大小

图 2-10 对话框中，用户可以自行决定是否安装 MariaDB 服务器的反馈插件。选择开启插件后，MariaDB 服务器的信息会定期上传到 mariadb.org 网站。上传的信息主要用于帮助厂商进一步改善 MariaDB 的功能，反馈插件收集的这些信息不包含涉及安全的敏感数据信息，主要是一些硬件相关信息，还包含 MariaDB 插件或功能的使用信息等。如果用户不希望上传这些信息，或者安装 MariaDB 服务器的硬件设备无法连接到外部网站时，请不要开启反馈插件。

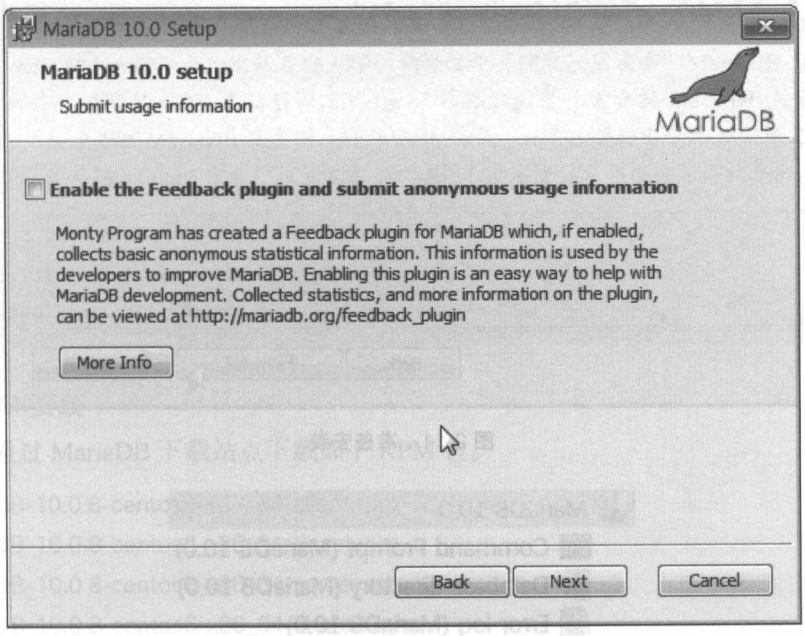


图 2-10 反馈插件安装选项

- MariaDB 服务器安装平台与硬件信息
- 安装 MariaDB 服务器的硬件所在国家信息
- MariaDB 服务器版本与编译信息
- 使用的功能与选项

所有安装完成后，Maria 10.0 菜单中出现用于访问 MySQL Client 与 MariaDB 设置文件（my.ini）的快捷菜单，如图 2-12 所示。

在 Windows 中安装完 MariaDB 10.0 后，MariaDB 的默认客户端“mysql”与名为“HeidiSQL”的 GUI 客户端也会同时安装。如果不习惯使用 mysql 客户端，可以选择使用 HeidiSQL 图形客户端。

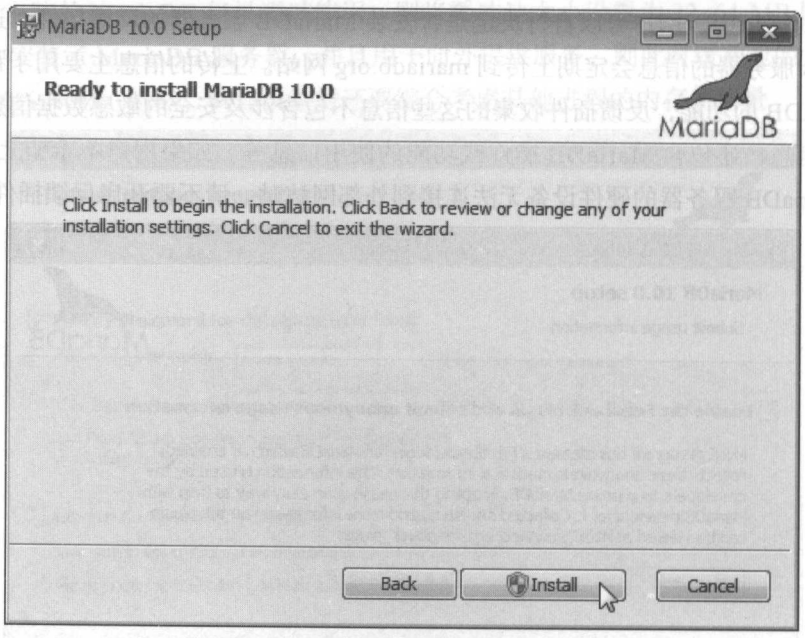


图 2-11 准备安装

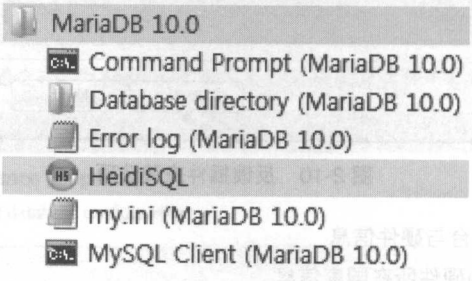


图 2-12 “开始”中的 MariaDB 快捷菜单

2. 安装压缩格式（ZIP）的 MariaDB

安装压缩格式的 MariaDB 非常简单，只要将下载的 ZIP 文件进行解压缩，然后将其移动到程序所在目录或 MariaDB 的专用目录即可。但是，这种安装方式默认不创建 MariaDB 的数据目录、默认数据库（mysql 数据库）以及设置文件（my.cnf），所以需要用户自己创建数据目录与设置文件。当然，创建过程并不复杂，但如果安装 MariaDB 只是为了进行简单的测试学习，建议各位还是使用 MSI Installer 进行安装。

通过解压缩安装好 MariaDB 后，用户可以根据需要自行调整 MariaDB 设置文件，方法与采用 MSI Installer 方式时是一样的。在 Windows 中解压并安装好 MariaDB 服务器后，它不会自动注册为 Windows 服务，需要另外编写启动脚本进行启动。

2.2.2 在 Linux 操作系统下安装 MariaDB

Linux 操作系统下的 MariaDB 安装方法有两种，一种是使用 RPM 包，另一种是使用 tar.gz 压缩包。若操作系统为 Red Hat 或 CentOS，则可以使用 YUM 等自动化包管理工具安装 MariaDB RPM 包，并进行相应更新。

注意 XtraDB 存储引擎是 MariaDB 5.5 的默认存储引擎，编译其中并随之一同发布。但是从 MariaDB 10.0 开始，Oracle 的 InnoDB 成为默认存储引擎，编译其中并随之一同发布。MariaDB 10.0.7 开始支持 XtraDB，用户若要使用 XtraDB 存储引擎而非 InnoDB 存储引擎，需要对 MariaDB 的设置文件（my.cnf）进行修改，然后重启 MariaDB 服务器。ha_xtradb.so 是 XtraDB 存储引擎的共享库文件，它存储于 MariaDB 发布版本的 lib/plugin/ 目录。相关内容会在 XtraDB 存储引擎部分进行详细介绍。

```
[mysqld]
ignore_builtin_innodb = ON
plugin-load = ha_xtradb.so
```

1. 使用 RPM 安装

首先，通过 MariaDB 下载站点下载如下 RPM 包。

- MariaDB-10.0.8-centos6-x86_64-client.rpm
- MariaDB-10.0.8-centos6-x86_64-devel.rpm
- MariaDB-10.0.8-centos6-x86_64-server.rpm
- MariaDB-10.0.8-centos6-x86_64-shared.rpm

若需要，也可以同时下载并安装包含 MariaDB 测试模块的“MariaDB-10.0.8-centos6-x86_64-test.rpm”。测试模块包中包含用于测试 MariaDB 性能的测试工具。所有包下载完成之后，按如下顺序逐个执行 RPM 包即可。

```
shell> RPM -ivh MariaDB-10.0.8-centos6-x86_64-client.rpm
shell> RPM -ivh MariaDB-10.0.8-centos6-x86_64-devel.rpm
shell> RPM -ivh MariaDB-10.0.8-centos6-x86_64-server.rpm
shell> RPM -ivh MariaDB-10.0.8-centos6-x86_64-shared.rpm
```

使用 RPM 安装 MariaDB 的过程中，若需要下面这些包含存储引擎库的包时，则要另行安装。

- MariaDB-10.0.8-centos6-x86_64-cassandra-engine.rpm
- MariaDB-10.0.8-centos6-x86_64-connect-engine.rpm
- MariaDB-10.0.8-centos6-x86_64-oqgraph-engine.rpm

RPM 安装完毕后，请务必修改 MariaDB 服务器的密码。

```
shell> mysqladmin -u root -h "127.0.01" -p 3306 password '密码';
```

以默认模式安装 MariaDB 服务器时，会一同创建用于查看的 test 数据库。该 test 数据库对所有用户开放全部权限，允许所有人进行连接，未登录的用户也可以以匿名方式访问 MariaDB 服务器。要想防止发生由此带来的安全问题，运行 mysql_secure_installation 安装脚本即可。

若所用的操作系统为 Red hat 或 CentOS，也可以使用 YUM 安装 MariaDB 包。关于 YUM 设置的详细内容，请参考 MariaDB 用户手册。

https://mariadb.com/kb/en/installing-mariadb-with-yum/

2. 安装压缩格式（tar.gz）的 MariaDB

Linux 操作系统下的压缩格式 MariaDB 安装方法与 Windows 下的方法几乎完全一致。如果各位想亲自创建与自己服务环境匹配的发布版本，可以在原压缩版本的基础上再次将其压缩为所需的形态，或创建 RPM 版本。

2.3 更新

从 MySQL 迁移到具有相同源码库的 MariaDB 时，并不需要先将数据库从 MySQL 中导出，然后再导入 MariaDB。因为 MariaDB 与具有相同源码库版本（版本号前两位）的 MySQL 之间拥有很好的兼容性。从表 1-5 所示的版本兼容性表格中可以看出，MySQL 5.1 与 MariaDB 5.2、5.3 相互兼容。从 MySQL 5.1 升级到 MariaDB 5.2 或 5.3 的过程中，并不需要单独进行数据转储与导入操作。但源码库版本发生变化时（5.1 → 5.2 或 5.3），则要使用 mysql_update 实用工具对认证与默认字典信息表（MySQL 服务器的 mysql 数据库中的表格）的结构进行升级。

从 MySQL 升级到 MariaDB 时，客户端的 JDBC 驱动与 MySQL C/C++ 客户端并不需要进行升级或编译。因为 MySQL 与 MariaDB 使用相同的客户端协议，相互间兼容。当然，若想再次发布客户端应用程序，那么使用 MariaDB 的 JDBC 驱动与 C/C++ API 要比使用 MySQL JDBC 与 C/C++ API 在许可协议上要更自由一些。

2.3.1 从 MySQL 升级到 MariaDB 时的注意事项

MySQL 处理查询的过程中需要临时表时，会使用 MEMORY 存储引擎与 MyISAM 存储引擎。因此，即使 MySQL 中并不主要使用 MyISAM 存储引擎，但也有必要对默认的 MyISAM 选项进行调优。与 MySQL 一样，MariaDB 处理查询时也会使用 MEMORY 存储引擎，但是对于创建在磁盘上的临时表，它会使用 Aria 存储引擎而非 MyISAM 存储引擎。因此，从 MySQL 升级为 MariaDB 时，要为 MyISAM 设置键缓冲区（key buffer）大小，还要为 Aria 存储引擎设置好页面缓存（page cache）大小。

```
aria-pagecache-buffer-size=64M
```

Aria 存储引擎的页面缓存类似于 MyISAM 存储引擎的 key_buffer_size，它表示对 Aria 存储引

擎的数据与索引页进行缓存的内存空间。但是 MyISAM 的键缓冲区仅用于缓存 MyISAM 表的索引，而 Aria 存储引擎的页面缓存不仅可以缓存索引，还可以缓存数据文件的页面。后面会详细介绍 Aria 存储引擎，此处只要记住，从 MySQL 升级到 MariaDB 时需要设置上面这些选项。

使用复制技术对主 / 从 MySQL 服务器分别进行升级时，也有一些注意事项。MariaDB 服务器通常能够识别 MySQL 服务器的二进制日志，但有时 MySQL 服务器不能识别 MariaDB 的二进制日志。因此，分别对主 / 从数据库服务器进行升级时，一定要先将从服务器升级为 MariaDB，然后再切换主 / 从服务器，或者将主 MySQL 服务器升级为 MariaDB。

此外，谁也无法保证升级过程中不会发生任何问题，所以开始升级或迁移前，一定要先对数据文件、设置文件等进行备份。这样，升级失败时也能顺利返回之前的 MySQL 服务器。

2.3.2 不受版本限制的安全升级法

从旧版 MySQL 升级为任一版本 MariaDB 的过程中，有一种安全的升级方法，使用户不必考虑版本差异的问题。使用该方法时，先使用 mysqldump 等逻辑备份工具转储所有数据，然后安装 MariaDB 服务器，再将备份好的数据加载其中。由于 mysqldump 使用标准的 ANSI SQL 语句转储数据，所以能够将利用其转储的数据顺利迁移到其他任何版本的数据库服务器。但是，使用 mysqldump 进行备份会相当耗时，而将转储的数据加载到新的 MariaDB 服务器耗时也比较长，所以并不适用于有海量数据进行迁移的情形。

(1) 使用 mysqldump 从旧版 MySQL 服务器中备份数据。

```
shell>mysqldump -u root -p --all-databases > backup.dump
```

(2) 安装并运行新版 MariaDB。

(3) 在新版 MariaDB 中执行由 mysqldump 备份的转储文件。

```
shell> mysql -u root -p < backup.dump
```

(4) 将转储数据全部加载到新版 MariaDB 后，执行 mysql_update 命令。

```
shell>mysql_upgrade --verbose
```

步骤 3 中进行数据导入时，如果是 Linux 或 Windows 环境，则可以通过在命令行中直接输入命令进行导入，也可以在打开的 MariaDB 客户端中使用 source 命令加载数据。我个人不太喜欢使用重定向导入转储文件的方式，更喜欢在 MariaDB 客户端中使用 source 命令导入数据。因为使用重定向方式导入转储文件时，画面上不会显示任何信息；而使用 source 命令时，处理细节会以转储文件的 SQL 语句为单位不断显示，这能够使我们知道导入操作在持续进行。

```
shell> mysql -h localhost -u root -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 3
Server version: 10.0.6-MariaDB-log mariadb.org binary distribution
```

```
Copyright (c) 2000, 2013, Oracle, Monty Program Ab and others.
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> source employees.sql
Query OK, 0 rows affected (0.29 sec)

Query OK, 0 rows affected (0.34 sec)

Query OK, 0 rows affected (0.47 sec)

Query OK, 0 rows affected (0.47 sec)
...
```

数据加载完毕后，最好再使用 mysql_upgrate 命令更新数据表。若转储文件仅包含服务数据，也可以不使用 mysql_upgrate 命令。但若其中包含之前版本的 MySQL 服务器中的所有数据库（含 mysql、information_schema 等用于管理的 DB），那么，由于 mysql DB 中的认证与字典表结构在不同版本间可能会不兼容，所以一定要使用 mysql_upgrate 命令将其升级到当前 MariaDB 版本。

```
shell> mysql_upgrade -h localhost -u root -p
Enter password:
Phase 1/3: Fixing table and database names
Phase 2/3: Checking and upgrading tables
Processing databases
information_schema
mysql
mysql.column_stats                                 OK
mysql.columns_priv                                 OK
mysql.db                                           OK
mysql.user                                         OK
performance_schema
...
Phase 3/3: Running 'mysql_fix_privilege_tables'...
OK
```

2.3.3 从 MySQL 5.0 或之前版本升级为 MariaDB 5.5

“从 MySQL 5.0 或之前版本升级为 MariaDB 5.5”不是指从之前版本的 MySQL 中转储数据，然后再将数据导入 MariaDB 5.5。它是在保持原有数据文件不变的前提下，只将 MySQL 服务器程序从 4.1 或 5.0 升级为 MariaDB 5.5。若想使用 mysqldump 更新 MySQL 服务器，请参考 2.3.2 节。

MySQL 5.0 或 4.1 与 MariaDB 5.1 的数据文件中，有如下几种显著变化。

- 索引列排序规则变化。

- MySQL 4.1 与 5.0 更改了 DECIMAL 与 VARCHAR 数据类型的存储方式。
- MySQL 5.1 开始采用编码存储，使数据表名在所有平台一致。

为了适应以上变化，就要使用 mysql_upgrade 实用工具对原有数据文件进行变换。若只将 MariaDB 程序升级为 5.5 版本，那么向外提供服务时就会出现无法读取部分数据表的问题，或者数据表名显示异常（#mysql50#...）。

(1) 备份原有数据（备份原版数据库以应对更新失败）；
(2) 保持原有数据不变的状态下删除旧版 MySQL 服务器；
(3) 安装 MariaDB 5.5 并调整设置文件；
(4) 启动 MariaDB 5.5；
(5) 运行 mysql_upgrade。

执行 mysql_upgrade 命令完成更新后，MariaDB 的数据目录中将生成名为“mysql_upgrade_info”的文件。该文件记录着执行 mysql_upgrade 命令的 MariaDB 服务器的版本，执行过 mysql_upgrade 命令的服务器或支持 mysql_upgrade 命令的 MariaDB 服务器中再次执行 mysql_upgrade 命令时，将会使用该文件中的信息，然后输出“更新已完成”的提示信息。若想再次强制执行 mysql_upgrade 命令，可以先删除“mysql_upgrade_info”文件再执行，或在执行 mysql_upgrade 命令时使用“--force”选项。

2.3.4 从 MySQL 5.1 升级为 MariaDB 5.5

“从 MySQL 5.1 升级为 MariaDB 5.5”不是指从之前版本的 MySQL 服务器中转储数据，然后再将数据导入 MariaDB 5.5。而是在保持原有数据文件不变的前提下，只将 MySQL 服务器程序从 5.1 升级为 MariaDB 5.5。其实，MySQL 5.1 与 MariaDB 5.5 之间的数据文件格式没有太大差异，所以升级时不会出现什么问题，升级方法与前面介绍的方法也并无区别。无论升级还是降级，开始之前都先要备份重要数据。

(1) 备份原有数据（备份原版数据库以应对更新失败）；
(2) 保持原有数据不变的状态下删除旧版 MySQL 服务器；
(3) 安装 MariaDB 5.5 并调整设置文件；
(4) 启动 MariaDB 5.5；
(5) 运行 mysql_upgrade。

我们暂且不谈从 MySQL 升级为 MariaDB 的问题。从 MySQL 5.1 升级为 MariaDB 5.5 或 MySQL 5.5 时，原设置文件中的“default-character-set”等设置选项会消失，它们会使 MariaDB 5.5 在启动时发生问题。出现问题时，仔细查看 MariaDB 的错误日志即可轻松把握究竟是哪些设置出了问题。在 MariaDB 5.5 或 MySQL 5.5 的用户手册中查找这些选项的设置，可以跟踪到消失的设置变量，或者查看它们是如何改变的。

无论版本差异多么小、变化多么细微，我都建议各位一定要运行 mysql_upgrade 程序。因为

即使不同版本间的数据文件格式或数据类型的存储格式一样，升级 MariaDB 版本时，mysql DB 中与用户或认证相关的表以及存储程序所保存的表列都会不断增加或发生变化。将 MySQL 5.1 升级为 MariaDB 5.5 后，有时即使不运行 mysql_upgrade 实用程序，MariaDB 服务器也能正常启动。但创建或删除主树或存储过程时，就会输出异常错误信息。因此，即使更新后暂时没有发生问题，也应当运行 mysql_upgrate 实用程序。

注意 由复制构成的 MariaDB 或 MySQL 服务器中，有时只在从服务器中执行 mysql_upgrade 命令，而未在主服务器中执行。在这些由复制构成的 MariaDB 或 MySQL 主服务器中运行 mysql_upgrade 命令时，为了更改（更新）mysql 数据库的 Schema，向从服务器传递执行的 DDL 语句时就会产生复制错误。此时，如果在主服务器中执行 mysql_upgrade 但并未在从服务器中检查其是否正常执行，就会发生以上错误，使复制停止并维持原状，这可能导致以后出现问题。因此，尽量不要将执行 mysql_upgrade 命令操作记录到二进制日志，运行时要使用"--skip-write-binlog" 选项，并且要在主 / 从服务器中分别运行。

```
shell>mysql_upgrade --force --skip-write-binlog
```

2.3.5 从 MySQL 5.5 升级为 MariaDB 5.5

MySQL 5.5 和 MariaDB 5.5 的数据格式与 mysql DB 的字典信息都是一样的，所以升级后不必再执行 mysql_upgrade 命令。即便执行 mysql_upgrade 命令也只是做些简单的检查，而不会转换文件。

从 MySQL 5.5 升级为 MariaDB 5.5 时，需要进行一些额外设置，以使 MariaDB 5.5 以最优方式启动。有关内容将在存储引擎与 MariaDB 系统设置部分进行详细讲解。

第3章

MariaDB 启动及执行查询

3.1 启动与终止服务器 MariaDB MySQL

3.2 登录服务器 MariaDB MySQL

3.3 创建数据库与用户 MariaDB MySQL

3.4 创建与修改数据表 MariaDB MySQL

3.5 数据操作 MariaDB MySQL

3.1 启动与终止服务器 MariaDB MySQL

在 Linux 中用 RPM 或在 Windows 中用 MSI 安装程序安装 MariaDB 服务器后，这两个平台下的 MariaDB 服务器都会自动注册为服务。MariaDB 服务器注册为服务后，根据操作系统的各启动级别，MariaDB 服务器会设置为自动启动或适时终止。与 MySQL 服务器一样，MariaDB 服务器注册为服务后，可以以守护进程模式（Daemon mode）启动；但是为了安装多个版本进行测试，也可以以非守护模式启动。下面讲解 MariaDB 服务器注册为服务后的启动与终止方法，并简单介绍不用作服务时的启动与终止方法。

3.1.1 在 Linux 中启动与终止 MariaDB 服务器

在 Linux 中将 MariaDB 服务器注册为服务，或者执行 mysqld_safe 脚本直接启动，这两种方法其实都不会对 MariaDB 服务器的运行方式产生影响。实际运行环境中，MariaDB 服务器通常会注册为服务，使 MariaDB 能够自动终止或启动。但是安装在个人 PC 或笔记本中用于测试学习时，人们一般不希望 MariaDB 服务器自动启动，有时需要亲自启动 MariaDB 服务器，此时可以使用 mysqld_safe 或 support-files/mysql.server 脚本。

1. MariaDB 注册为服务时

在 Linux 系统中以 RPM 方式安装好 MariaDB 服务器后，名为 mysql 的 MariaDB 服务器初始化脚本也会一同安装到 /etc/init.d/ 目录。并且，MariaDB 自动注册为服务，这使 MariaDB 服务器能够根据相应的初始化级别（Linux init 级别）自动终止并再次重启。

MariaDB 服务器注册为服务后，启动与终止的方法相当简单。

```
## 启动 MariaDB 服务器
shell> service mysql start

## 终止 MariaDB 服务器
shell> service mysql stop
```

此外，MariaDB 注册为服务后，可以使用如下命令查看 MariaDB 服务器的进程状态。

```
shell> service mysql status
MySQL running( 19807 )   [OK]
```

执行以上命令后，返回 MySQL 的进程号为 19807。由此可知，该进程正常启动。若 MariaDB 服务器进程未启动，则会输出“MySQL is not running[Failed]”信息。

若不采用 RPM 方式安装 MariaDB 而直接编译源代码，或者通过 tar.gz 压缩版本安装 MariaDB，则 MariaDB 的初始化脚本不会自动安装到 /etc/init.d/ 目录。此时，只需要从 MariaDB 安装目录下的 support-files 文件夹中名为 mysql.server 的脚本文件复制到 /etc/init.d/ 目录即可。下列示例中，MariaDB 安装到 /usr/local/mysql 目录，使用 cp 命令将 mysql.server 脚本文件复制

到 /etc/init.d/ 目录，并修改名称为 mysql；执行 chkconfig 命令，根据 mysql 初始化脚本文件的头部分的定义，确定 mysql 服务器的启动与终止级别，然后将其注册为服务。

```
shell> cp /usr/local/mysql/support-files/mysql.server /etc/init.d/mysql
shell> chkconfig mysql
```

如上所示，执行 chkconfig 命令后，即可使用“service mysql start”命令或“service mysql stop”命令启动或终止 MariaDB 服务器，就像采用 RPM 方式安装 MariaDB 后的启动或终止一样。

2. MariaDB 不注册为服务时

有时需要在系统中安装多个版本的 MariaDB 以进行测试。此时如果再将 MariaDB 注册为服务，管理起来就会比较麻烦。由于多个 MariaDB 无法同时使用同一端口，所以在 Linux 中以服务形式运行会使管理变得很麻烦。特别是在个人 PC 中，每次重启都要启动 MariaDB 服务器，这对资源是很大的浪费。

此情形下，不要将 MariaDB 注册为服务，而是可以像下面这样创建简单的脚本，每次需要时手动启动 MariaDB 服务器。

```
#!/bin/bash

MYSQL_HOME=/usr/local/mysql
${MYSQL_HOME}/bin/mysqld_safe --defaults-file=/etc/my.cnf &
```

若不想使用 mysqld_safe 脚本启动 MariaDB 服务器，那么只要在创建脚本时将 mysqld_safe 换成 mysqld 即可。此外，使用 support-files/mysql.server 脚本可以直接启动或终止 MariaDB 服务器。

```
## 启动
shell> sh ${MYSQL_HOME}/support-files/mysql.server start

## 终止
shell> sh ${MYSQL_HOME}/support-files/mysql.server stop
```

手动启动 MariaDB 服务器后，也可以使用 mysqladmin 实用工具终止 MariaDB 服务器。

```
shell> mysqladmin -u root -p shutdown
```

当然，将 MariaDB 注册为服务并使用“service mysql start”启动后，也可以像上面这样使用 mysqladmin 命令终止 MariaDB 服务器。

如果 MariaDB 服务器在其他服务器中启动或无法访问 Unix 套接字文件，那么使用 mysqladmin 命令关闭时要同时输入主机名、端口号，如下所示。如果要关闭的 MariaDB 服务器位于远程设备，则 root 用户必须能够从当前机器连接到 MariaDB 服务器。

```
shell> mysqladmin -h 192.168.0.1 -P 3307 -u root -p shutdown
```

MariaDB 服务器的 InnoDB 终止后，再次启动时，系统会利用事务日志自动恢复。但是，使用“kill -9”关闭 MariaDB 服务器仍然不是一个明智之举。请尽量使用 mysqladmin 或 service

mysql stop 命令关闭 MariaDB 服务器。

3.1.2 在 Windows 系统中启动与终止 MariaDB 服务器

Windows 系统中手动启动、关闭 MariaDB 服务器的方法与 Linux 系列操作系统中的几乎一样，但若 MariaDB 服务器注册为 Windows 服务，启动或关闭 MariaDB 服务器时就要使用 Windows 的“服务”面板了。

1. MariaDB 注册为服务时

在 Windows 系统中使用 MSI 安装程序安装 MariaDB 服务器的过程中，其中一个安装步骤要求用户输入服务名。MariaDB 服务器以服务形式安装后将使用该名称，即此处输入的服务名将出现在 Windows 服务管理器中。从图 3-1 可以看到，MariaDB 服务器注册为服务，服务名为“MariaDB 10.0.6”。

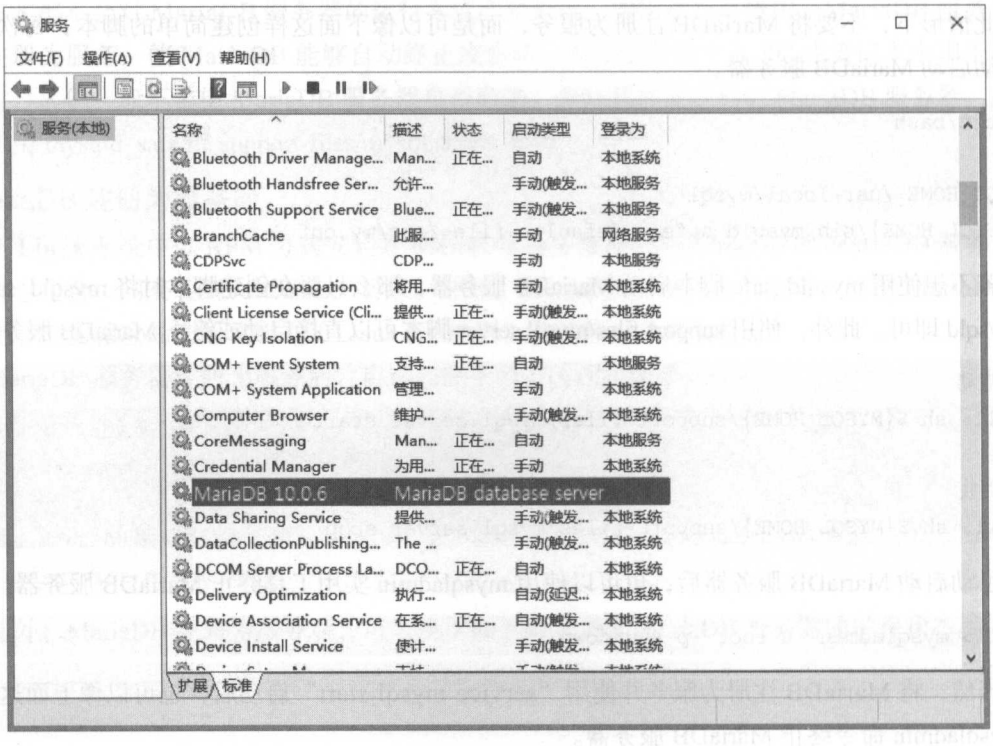


图 3-1 MariaDB 服务器注册为服务后的启动与终止

将 MariaDB 服务器注册为服务后，在服务管理窗口的工具栏中，单击“启动服务”或“停止服务”按钮，可以轻松启动或关闭 MariaDB 服务器。

2. MariaDB 不注册为服务时

Windows 系统中，不使用安装程序安装 MariaDB 服务器时，MariaDB 将不会自动注册为 Windows 服务。此时，与 Linux 系统一样，需要单独创建 MariaDB 启动脚本（bat 文件）启动

MariaDB 服务器。

Windows 版本的 MariaDB 中不存在单独的 mysqld_safe 程序，需要使用“—defaults-file”选项启动 mysqld 程序。

```
c:\mysql\MariaDB10.0.6\bin\mysqld --defaults-file=c:\mysql\MariaDB10.0.6\my.ini
```

与 Linux 系统一样，关闭 MariaDB 服务器时也要使用 mysqladmin 程序。

```
c:\mysql\MariaDB10.0.6\bin\mysqladmin -h127.0.0.1 -P3306 -u root -p shutdown
```

3.2 登录服务器 MariaDB MySQL

连接 MariaDB 服务器时，可以使用 mysql 命令行工具，也可以使用随 MariaDB 安装程序（MSI）一起安装的 HeidiSQL GUI 工具。

3.2.1 登录服务器与版本确认

运行 mysql 命令行工具后，出现“Enter password:”提示信息，等待用户输入密码。用户输入正确密码后，出现 MariaDB 提示符，等待用户进一步输入访问命令。

```
C:\mysql> c:\mysql\MariaDB10.0.6\bin\mysql -h 127.0.0.1 -P 3306 -u root -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 3
Server version: 10.0.6-MariaDB-log mariadb.org binary distribution

Copyright (c) 2000, 2013, Oracle, Monty Program Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]>
```

若所连接的 MariaDB 服务器位于本地计算机，则可以省略“-h 127.0.0.1”选项。MariaDB 服务器使用 MySQL 默认的 3306 端口时，“-p 3306”选项也可以省略。上述示例使用 mysql 客户端连接 MariaDB 服务器，mysql 客户端随 MariaDB 一起安装到系统。连接成功后，从返回的信息中可以看到，“Server version:”为“10.0.6-MariaDB-log”。

下面尝试使用 MySQL 的 mysql 客户端连接同一 MariaDB 服务器，并查看返回结果。

```
C:\mysql>c:\mysql\mysql5.6.15\bin\mysql -h 127.0.0.1 -P 3306 -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3
```

```
Server version: 5.5.5-10.0.6-MariaDB-log mariadb.org binary distribution

Copyright (c) 2000, 2013, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

如上所示，使用 MySQL（非 MariaDB）安装程序包含的 mysql 客户端连接到 MariaDB 服务器后，从返回的信息中可以看到，“Server version:”为“5.5.5-10.0.5-MariaDB-log”。这是为了保证 MariaDB 与 MySQL 的 mysql 客户端程序之间的版本兼容性，所以 MariaDB 服务器对版本进行了一些修改后输出。也就是说，MySQL 中的 mysql 客户端程序无法识别 MariaDB 10.0.x 版本，只输出兼容的 MariaDB 5.5.5 版本号，而其后 MariaDB 服务器实际的版本号为 10.0.6。

由于 mysql 客户端程序先使用该信息判断服务器识别哪个版本的语法、使用哪个版本的协议，然后再与服务器进行通信，所以为了保证兼容性，MariaDB 服务器会根据客户端的种类返回不同的版本信息。

若想查看与 MariaDB 服务器连接相关的信息，可以在 mysql 客户端程序中输入“\s”或“status”命令。该命令非常有用，执行之后，MariaDB 会返回当前服务器的版本、客户端连接账号，以及字符集相关信息等。

```
MariaDB [(none)]> \s
--------------
c:\mysql\MariaDB10.0.6\bin\mysql  Ver 15.1 Distrib 10.0.6-MariaDB, for Win32 (x86)

Connection id:          4
Current database:
Current user:           root@localhost
SSL:                    Not in use
Using delimiter:        ;
Server:                 MariaDB
Server version:         10.0.6-MariaDB-log mariadb.org binary distribution
Protocol version:       10
Connection:             127.0.0.1 via TCP/IP
Server characterset:    utf8
Db     characterset:    utf8
Client characterset:    utf8
Conn.  characterset:    utf8
```

```
TCP port:               3310
Uptime:                 2 min 28 sec

Threads: 2  Questions: 7  Slow queries: 0  Opens: 0  Flush tables: 1  Open
tables: 63  Queries per second avg: 0.047
--------------
```

3.2.2 mysql 客户端程序支持的选项

mysql 客户端程序没有漂亮的 GUI，但是它提供了丰富的功能，以及大量有用的工具。像 Linux 这类非 GUI 的终端环境中，mysql 命令行工具是非常有用的。它提供了大量可用的选项，用户输入不同的参数选项可以快速查询服务器中的数据。

在 Windows 命令行提示符中输入“mysql --help”命令，列出 mysql 客户端程序可以识别的所有选项。

```
C:\> mysql --help
c:\mysql\mysql5.6.15\bin\mysql  Ver 14.14 Distrib 5.6.15, for Win32 (x86)
Copyright (c) 2000, 2013, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Usage: c:\mysql\mysql5.6.15\bin\mysql [OPTIONS] [database]
  -?, --help          Display this help and exit.
  -I, --help          Synonym for -?
  --auto-rehash       Enable automatic rehashing. One doesn't need to use
                      'rehash' to get table and field completion, but startup
                      and reconnecting may take a longer time. Disable with
                      --disable-auto-rehash.
                      (Defaults to on; use --skip-auto-rehash to disable.)
...
```

mysql 客户端支持的选项很多，下面只介绍其中几个常用选项。

- --host、-h：指定要连接的 MariaDB 服务器所在的主机地址。
- --port、-p：指定要连接的 MariaDB 服务器所在的端口号。
- --user、-u：连接 MariaDB 服务器时要用的用户名。
- --password、-p：连接 MariaDB 服务器时要用的密码（若省略 -p 选项值，则在运行 mysql 程序后自动出现输入提示要求输入密码）。
- --database、-D：mysql 客户端在连接 MariaDB 服务器时同时指定要使用的数据库。
- --auto-rehash：在 mysql 客户端程序内输入表或列名时，使用 TAB 键可以自动补全。

- --batch：以批处理模式（非交换模式）运行 mysql 客户端程序。
- --execute、-e：mysql 客户端程序在连接 MariaDB 服务器的同时执行参数给出的语句，属于非交互式模式。
- --skip-column-names、-N：在 mysql 客户端中不显示查询结果中的列名。
- --safe-updates、-U：以安全模式运行 mysql 客户端。安全模式下，SELECT、UPDATE、DELETE 查询不能使用索引并进行全表扫描时，查询会自动停止。操作重要的数据表时，使用安全模式可以有效防止误操作。

1. 以安全模式运行 mysql 客户端

安全模式下，SELECT、UPDATE、DELETE 查询（除 INSERT 外）不能使用索引时，MariaDB 服务器会产生如下错误，并终止查询。使用安全模式可以有效防止用户误操作，比如更新数据表的所有记录或全部删除。

```
MariaDB> CREATE TABLE tab_important (uid INT, uname VARCHAR(20), PRIMARY KEY(uid));
MariaDB> INSERT INTO tab_important VALUES (1, 'Matt');

MariaDB> DELETE FROM tab_important WHERE uname='Matt';
ERROR 1175 (HY000): You are using safe update mode and you tried to update a table
without a WHERE that uses a KEY column

MariaDB> UPDATE tab_important SET uname='Lee';
ERROR 1175 (HY000): You are using safe update mode and you tried to update a table
without a WHERE that uses a KEY column
```

2. 使用 --execute 选项运行 mysql 客户端

通过 --execute 选项可以设置要运行的查询语句，mysql 客户端连接 MariaDB 服务器时会将指定查询语句以参数形式传递，返回的查询结果将发送到标准输出（stdout）。

以非交互模式（非交互模式类似于 Linux 中的 cron 或 Windows 中的计划任务）运行 mysql 客户端程序时，*--execute 选项会非常有用*。

```
shell> mysql -h 127.0.0.1 -P 3306 -u root -p -D test --execute="SELECT * FROM tab_important"

+-----+-------+
| uid | uname |
+-----+-------+
|   1 | Matt  |
+-----+-------+
```

3. 同时使用 --batch 与 --execute 选项运行 mysql 客户端

默认情况下，mysql 客户端以非批处理模式运行，查询结果会以文本模式的表形态输出列名及值。以 --batch 模式运行时，查询结果将以无线表格形式输出。需要使用 shell 脚本操作查

询的运行结果时，使用 --batch 模式会比较好。

```
shell> mysql -h 127.0.0.1 -P 3306 -u root -p -D test --batch
               --execute="SELECT * FROM tab_important"

uid     uname
1       Matt
```

4. 同时使用 --batch、--skip-column-names、--execute 选项运行 mysql 客户端

想在 shell 脚本中操作查询的运行结果时，同时使用 --batch 与 --skip-column-names 选项可以将 shell 脚本中不必要的列名删除，仅将列值显示到标准输出。shell 脚本要从 MariaDB 服务器读取某些值并进行某种处理时，可以同时使用这些选项。

```
shell> mysql -h 127.0.0.1 -P 3306 -u root -p -D test --batch --skip-column-names
             --execute="SELECT * FROM tab_important"

1       Matt
```

3.3 创建数据库与用户 MariaDB MySQL

安装 MariaDB 服务器时，会自动或通过 mysql_install_db 实用工具创建几个默认的数据库与表。这些数据库是启动 MariaDB 服务器时必备的，启动之前要先行创建。此外，还会与默认数据库一起创建第一次连接 MariaDB 服务器时要使用的用户账户。下面看看这些默认创建的数据库、表以及用户，然后学习自己创建数据库与用户。

3.3.1 MariaDB 用户账户识别与权限

MySQL 和 MariaDB 的用户信息与其他 RDBMS 略有不同。连接 MariaDB 服务器时，除用户名称外，用户用于连接的客户端也是认证的一部分。请各位注意这一点，否则连接 MariaDB 服务器时可能引发各种问题。

1. 用户识别

与其他 DBMS 略微不同，MariaDB 的用户账号不仅包含用户名，而且用户的连接点（运行客户端的主机名、域名、IP 地址）也是用户账号的组成部分。因此，谈及 MariaDB 中的用户账号时，总是包括用户的 ID 与主机两个部分。用户 ID 与 IP 地址要分别用反引号（`）区分开来，在 MariaDB 中用于包围标识符，常常使用单引号（'）表示。下面的用户账号只能在运行 MariaDB 服务器的机器中使用 svc_id 连接 MariaDB 服务器。若只注册了下面一个用户账号，那么在其他计算机中使用 svc_id 将无法连接到 MariaDB 服务器。

```
'svc_id'@'127.0.0.1'
```

若想创建可以从外部所有计算机连接 MariaDB 服务器的用户账号，只需将用户账号的主机部分替换为“%”即可，“%”代表所有 IP 或主机名。

如果存在多个重复账号，MariaDB 服务器进行用户认证时会从中选取一个。此时，MariaDB 服务器会选择哪一个呢？假如 MariaDB 服务器中存在下面两个重复的账号。

```
'svc_id'@'192.168.0.10'（该账号密码为 123）
'svc_id'@'%'            （该账号密码为 abc）
```

在 IP 地址为 192.168.0.10 的 PC 上连接 MariaDB 服务器时，连接成功与否取决于用户使用第一个还是第二个账号信息进行连接。MariaDB 服务器会选择哪一个呢？选择权限与账号信息时，MariaDB 总是先选择范围最小的那个。上面两个账号信息中，'svc_id'@'192.168.0.10' 范围比 'svc_id'@'%' 小，所以 MariaDB 会选用带有明确 IP 地址信息的用户账号进行认证。用户在 IP 地址为 192.168.0.10 的 PC 上使用 'svc_id' 用户名与 abc 密码进行登录时，会因密码不一致而遭到 MariaDB 服务器拒绝。那么，为什么 MariaDB 服务器允许重复账号存在呢？因为数据库系统的运维过程中常常会遇到这种需求。但是，这也会引发一些问题，比如开发者常常会遇到“无法连接 MariaDB 服务器”的情况，复制过程中还经常发生无法从 MariaDB 服务器连接主 MySQL 服务器等诸多问题。

2. 权限

MariaDB 服务器中存在多种权限（Privileges），只有理解这些基本权限，才能根据用途为连接服务器的用户赋予相应权限。从 MariaDB 10.0 开始，用户可以创建权限组（Role），并可以将它赋予某特定用户，一个权限组包含多个权限。

根据 MariaDB 权限的作用范围，可以将权限分为服务器级权限、数据库级权限、表级权限，以及与存储程序相关的权限。此外还有列级权限，由于这种权限并不常用，故省略。为某个用户赋予权限时，要使用如下所示的 GRANT 命令。其中，ON 子句用于指定数据库对象，可指定的数据库对象有如下几种。

(1)*.*

该对象为整个 MariaDB 服务器，此时在 privilege_list 中只能用全局权限。

(2) db1.

该对象为某个特定的数据库，privilege_list 中可以出现除全局权限外的数据库级、表级、存储程序级权限。

(3) db1.table1

该对象为某个数据库中的一个表，privilege_list 中只能使用表级权限。

(4) db1.stored_program1

该对象为某个数据库的一个存储程序，privilege_list 中只能使用存储程序级别权限。

```
GRANT privilege_list ON db.TABLE TO 'user'@'host' ;
```

下面介绍 MariaDB 中一些作用范围不同的权限。

全局权限

权限	说明
CREATE USER	创建一个新账户的权限
FILE	使用 LOAD DATA INFILE 或 LOAD_FILE() 函数访问磁盘文件时所需的权限
GRANT OPTION	可以对其他用户授权的权限（即使有 GRANT 权限，也只能将自己拥有的权限授予其他用户）
PROCESS	该权限下，使用 SHOW PROCESSLIST 命令能够查看 MariaDB 服务器内的进程
RELOAD	拥有该权限即可执行 FLUSH…命令
REPLICATION CLIENT	拥有该权限后，使用 SHOW MASTER STATUS 与 SHOW SLAVE STATUS 命令可以查看主 / 从服务器的状态
REPLICATION SLAVE	从 MariaDB 服务器连接主 MariaDB 服务器时，使用的用户账号需要该权限（只有拥有该权限，从服务器才能访问主 MariaDB 服务器中发生的更改）
SHOW DATABASES	显示数据库列表的权限
SHUTDOWN	关闭 MariaDB 服务器的权限
SUPER	请参考下文

SUPER 权限与 Unix 中的 root 用户权限含义不同，拥有该权限后，在某种特定情形下可以突破某些限制并完成操作。比如，设置为 read_only 的 MySQL 服务器可以使用 SUPER 权限修改数据；连接达到 max_connections 上限而无法再创建连接时，拥有 SUPER 权限的用户可以再创建一个连接。MySQL 5.0（该版本中无单独定义的 TRIGGER 权限）中，创建或删除触发器时也需要拥有 SUPER 权限。各位可能有时感觉 SUPER 权限不够明确，某种特定情形下需要的权限都会分配 SUPER 权限。因此，尽量不要为普通用户或服务账号授予 SUPER 权限。

数据库级权限

权限	说明
CREATE	创建新数据库的权限
CREATE ROUTINE	创建新存储程序或函数的权限
CREATE TEMPORARY TABLES	创建临时表的权限（与创建内部临时表无关）
DROP	删除数据库的权限
EVENT	创建、删除事件的权限
GRANT OPTION	拥有该权限后，可以将数据库权限赋予其他用户（向其他用户授权时，仅限于自己拥有的权限）
LOCK TABLES	拥有该权限后，可以使用 LOCK TABLES 命令锁定表

表级权限

权限	说明
ALTER	修改表结构的权限
CREATE	创建表的权限
CREATE VIEW	创建视图的权限
DELETE	删除（DELETE 查询）表记录的权限

（续）

权限	说明
DROP	删除表或视图的权限
GRANT OPTION	拥有该权限即可将表权限授予其他用户
INDEX	拥有该权限后，可以使用 CREATE INDEX 命令创建索引（若拥有创建表的权限，则即使无该权限，也可以随 CREATE TABLE 语句一起创建索引）
INSERT	向表中存储（INSERT 查询）记录的权限
SELECT	访问（SELECT 查询）表记录的权限
SHOW VIEW	拥有该权限后，使用 SHOW CREATE VIEW 命令可以查看视图结构
TRIGGER	创建、删除、执行触发器的权限
UPDATE	修改（UPDATE 查询）表记录的权限

存储程序级权限

权限	说明
ALTER ROUTINE	修改存储程序（过程与函数）内容的权限
EXECUTE	执行存储程序（过程与函数）的权限
GRANT OPTION	拥有该权限后，可以向其他用户授予存储程序（过程与函数）权限

3. 赋予权限

为用户赋予权限时，要使用 GRANT SQL 语句。GRANT 语句使用如下语法编写，编写时要根据各权限的作用范围指定相应的数据库对象。

```
mariadb> GRANT privilege_list ON db.TABLE TO 'user'@'host' ;
mariadb> GRANT privilege_list ON db.TABLE TO 'user'@'host' identified BY 'password'
         WITH GRANT OPTION ;
```

使用 GRANT 语句可以为某个特定用户分配权限，若指定用户不存在，则先创建用户，然后再为其赋予权限。指定用户尚未创建时，使用 GRANT 语句可以同时创建用户与赋予权限。并且，在 GRANT 命令中使用 IDENTIFIED BY 子句可以为创建的用户设置密码。与其他权限不同，GRANT OPTION 权限（为其他用户授权的权限）在 GRANT 语句的末尾使用 WITH GRANT OPTION 子句赋予。privilege_list 中同时有多个权限时，可以使用逗号（,）将多个权限分隔开。TO 关键字之后是获得权限的用户，ON 关键字之后用于指定向哪个数据库的哪个对象授予权限。权限范围不同，所用的方法也不相同。下面通过简单的示例进行介绍。

(1) 全局权限

```
mariadb> GRANT SUPER ON *.* TO 'user'@'localhost';
```

授予全局权限时，在 ON 子句中使用“*.*”，表示该权限的作用范围为所有 DB 的所有对象。也就是说，由于无法只针对某个特定 DB 授予全局权限，所以授予全局权限时不能将其作用范围指定为某个 DB 或特定对象。SUPER 这类全局权限不能以 DB 或数据对象为单位进行授予，所以授予时一般只使用“*.*”。

(2) 数据库权限

```
mariadb> GRANT EVENT ON *.* TO 'user'@'localhost';
mariadb> GRANT EVEMT ON employees.* TO 'user'@'localhost';
```

授予 DB 权限时，可以只针对某个特定的 DB，也可以针对服务器中的所有 DB。因此，授权时在 ON 子句中既可以使用“*.*”，也可以使用“employees.*”，如上面示例所示。

但是授予 DB 权限时（由于授予 DB 权限时不能针对数据表），不能像 employees.department 一样连数据表也表示出来。由于 DB 权限可以应用于整个服务器的所有 DB，所以向指定用户赋予时可以使用“*.*”。此外，由于 DB 权限也可以只针对某个特定 DB，所以向指定用户授予该权限时，也可以使用“db.*”。但是不能像赋予对象权限一样连对象（数据表）也表示出来，比如“db.table”。

(3) 数据表权限

```
mariadb> GRANT SELECT,INSERT,UPDATE,DELETE ON *.* TO 'user'@'localhost';
mariadb> GRANT SELECT,INSERT,UPDATE,DELETE ON employees.* TO 'user'@'localhost';
mariadb> GRANT SELECT,INSERT,UPDATE,DELETE ON employees.department TO 'user'@'localhost';
```

如第一个示例所示，授予对象权限时，“对象”可以指服务器里所有数据库中的对象；也可以仅指某个特定 DB 的对象，如第二个示例所示。当然，授权时也可以只针对某个特定 DB 中的特定表，如第三个示例所示。授予对象权限时，可以使用“*.*”，使权限作用于整个服务器中的所有对象。由于权限也可以应用到特定 DB 中的所有对象，所以在 ON 子句中也可以使用“db.*”。

只针对数据表的某个特定列授予权限时，GRANT 语句语法虽然与授予对象权限类似，但也有一些不同。可以向数据列赋予的权限有三种（不包含 DELETE），分别为 INSERT、UPDATE、SELECT。授权时，要在各权限后明确给出数据列名。下面的 GRANT 授权语句中，UPDATE 权限仅授给 employees DB 的 department 数据表中的 dept_name 列，而 SELECT、INSERT 操作可以作用于所有数据列。

```
mariadb> GRANT SELECT,INSERT,UPDATE (dept_name) ON employees.department TO
'user'@'localhost';
```

设置权限时虽然可以为权限指定多个级别、多个范围，但是表级权限与列级权限并不常用。设置一个数据列权限后，数据库系统会对所有表的所有列进行权限检查。这意味着，即使只为一个数据列设置权限，也会影响整个数据库系统的性能。如果确实需要为某个数据列设置访问权限，建议不要使用 GRANT 命令，而是将需要设置权限的数据列从相应数据表中抽取出来，建立视图（VIEW），然后再通过视图进行相应的访问控制。

使用 GRANT 命令赋予的权限信息存储于 mysql 数据库（MariaDB 的认证与字典数据库）的 user 表与 host 表。

4. 权限组 MariaDB

从 MariaDB 10.0.5 开始，权限管理支持多种方式。不仅可以将各权限分别分配给用户，还可以将一些权限集中起来创建为一个权限组（Role），然后将权限组赋予个别用户。

(1) 创建权限组

用户可以使用 CREATE ROLE 命令创建权限组，但只有拥有 CREATE USER 权限的用户才能使用 CREATE ROLE 命令创建权限组。每次执行 CREATE ROLE 命令时，权限组的信息都会存储到 mysql 数据库的 user 表，并且 mysql.user 表的 is_role 列设置为“Y”。也就是说，mysql.user 表中同时存储着权限组与用户信息。

```
mariadb> CREATE ROLE dba;
mariadb> CREATE ROLE developer WITH ADMIN {CURRENT_USER | CURRENT_ROLE | user | role}
```

创建权限组时，要同时将可以把权限组赋予其他用户的用户明确指出来。第一个示例中，创建权限组时虽然省略了 WITH ADMIN 子句，但是数据库系统会自动添加默认的 WITH ADMIN CURRENT_USER。也就是说，只有创建 dba 权限组的用户才能将 dba 权限组授予其他用户。第二个示例中，创建权限组时使用了 WITH ADMIN 子句，并且在子句后面明确指出了 CURRENT_USER 或 CURRENT_ROLE，表示创建当前权限组的用户及该用户所属权限组的用户可以向其他用户授予该权限组。也可以在 WITH ADMIN 子句中直接指定特定用户或特定组。请注意，权限组名称不能用 PUBLIC 与 NONE，因为它们是保留字。

权限组创建完成后，可以像为普通用户授予权限一样，将相关权限授予特定权限组，如下列示例所示。

```
mariadb> GRANT ALL PRIVILEGES ON *.* TO dba;
mariadb> GRANT SELECT, INSERT, UPDATE, DELETE ON 'db1'.* TO developer;
```

(2) 授予权限组

使用 GRANT 命令可以将创建好的权限组授予特定用户。授予后，各用户与权限组间的关系信息就会存储到 mysql.roles_mapping 表。

```
mariadb> GRANT developer TO 'user1'@'%';
```

令人遗憾的是，MariaDB 10.0（当前最新版本为 MariaDB 10.0.8）的权限组无法自动识别赋予用户的权限组。

```
mariadb> SELECT CURRENT_ROLE;
+--------------+
| current_role |
+--------------+
| NULL         |
+--------------+
```

登录 MariaDB 服务器后，查看当前登录的用户权限组，可以发现没有任何设置。要想获得用户当前连接（会话）中的权限组，需要执行 SET ROLE 命令。

```
mariadb> SET ROLE dba;
Query OK, 0 rows affected (0.00 sec)

mariadb> SELECT CURRENT_ROLE;
+--------------+
| current_role |
+--------------+
| dba          |
+--------------+
```

还有一点令人遗憾的是，一个用户无法同时拥有两个以上的权限组。拥有 dba 权限组的状态下，再次执行 SET ROLE 命令，只能识别最后设置的权限组。

```
mariadb> SET ROLE developer;
Query OK, 0 rows affected (0.00 sec)

mariadb> SELECT CURRENT_ROLE;
+--------------+
| current_role |
+--------------+
| developer    |
+--------------+
```

删除设置的权限组时，只要将权限组设置为 NONE 即可，如下所示。

```
mariadb> SET ROLE NONE;
Query OK, 0 rows affected (0.00 sec)

mariadb> SELECT CURRENT_ROLE;
+--------------+
| CURRENT_ROLE |
+--------------+
| NULL         |
+--------------+
```

权限组功能从 MariaDB 10.0.5 开始出现，相信以后会不断改善。但就目前情形看，实际工作中使用权限组还是有些不便。希望在不久的将来，MariaDB 的权限组功能可以变得像 Oracle RDDMS 一样强大。

3.3.2 MariaDB 默认用户

在 Windows 中安装好 MariaDB 后，MariaDB 的主目录中会自动创建名为 data 的文件夹，

该文件夹保存着MariaDB服务器的数据文件。在Linux系列操作系统中，使用mysql_install_db命令即可生成默认数据。随MariaDB的安装而一起生成的数据文件中，默认有几个MariaDB服务器的用户账号。其实这些账号仅在第一次连接MariaDB服务器时使用，实际使用中往往会基于安全考虑而将这些账号删除，然后使用其他名称创建新账号。

- `root`@`127.0.1`
- `root`@`::1`
- ``@`localhost`
- `root`@`localhost`

用户账号的主机部分中，“127.0.0.1”与“localhost”代表的含义其实是一样的。但是，明确以localhost用户连接MariaDB服务器时，必须在-h选项中使用localhost才行。由于Windows中不支持使用套接字文件进行连接，所以“127.0.0.1”与“localhost”没什么不同。但是，如果MariaDB安装在Unix系统中，那么连接（不使用TCP/IP）安装于本地机的MariaDB服务器时，则可以使用Unix Domain Socket进行。此时若将主机名指定为localhost，则会自动使用Unix Domain Socket文件进行连接。在安装于本地计算机的MariaDB服务器中创建用户时，若有可能，请同时创建localhost与127.0.0.1两个用户账号。

MariaDB中，root用户账号初始化后默认拥有所有权限。初次安装的MariaDB服务器中，用户可以使用root账号登录服务器，且不需要输入密码；也可以匿名方式登录服务器，而不必给出任何用户名。因此，安装新MariaDB服务器后，总要先删除所有用户账号，再重新创建要使用的账号。人们通常认为MariaDB的管理员账号就是root账号，因为安装MariaDB服务器的早期root账号拥有所有权限。MariaDB中，名为root的账号并不总是必需的，需要时也可以将MariaDB服务器的管理员账号更改为admin（但是作为管理者所需的所有权限要另外进行授予）。若想从早期创建数据库时就解决这样的安全问题，使用mysql_secure_installation实用工具即可。

3.3.3 MariaDB的默认数据库

运行mysql客户端程序，如下所示登录MariaDB服务器后，执行“SHOW DATABASE;”命令即可查看服务器中的数据库列表。初次安装并运行MariaDB服务器时，存在名为root的DBA（数据库管理员）账号，我们会一直使用该root登录MariaDB服务器。

```
shell> mysql -h 127.0.0.1 -P3310 -u root -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 3
Server version: 10.0.6-MariaDB-log mariadb.org binary distribution

Copyright (c) 2000, 2013, Oracle, Monty Program Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
```

```
mariadb> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| test               |
+--------------------+
```

默认创建的数据库中，mysql 数据库存储着 MariaDB 服务器用户认证信息、存储程序与事件信息，以及与时区、复制相关的信息。因此，只有管理员才能访问 mysql 数据库，除此之外的其他用户都不能访问。若可能，要禁止用户直接修改其中的数据。

information_schema 数据库中的数据表不在物理磁盘上拥有实际数据文件。MariaDB 服务器启动时，所有数据库、数据表、数据列的信息以及存储程序的元信息都会读入内存，通过 information_schema 数据库中的各种数据表即可访问这些信息。灵活运用 information_schema 数据库中的数据表可以轻松查看 MariaDB 服务器的 schema 信息、存储程序信息以及索引统计信息等。

performance_schema 数据库中的数据表以数字形式记录着 MariaDB 服务器处理查询请求时发生的各种事件、锁、锁等待现象等。performance_schema 数据库中的数据表只在磁盘上存储表结构，实际数据都在内存中管理。MariaDB 服务器重启时，全部初始化。performance_schema 数据库中的数据表作用类似于 Oracle RDBMS 的 OWI（Oracle Wait Interface），它们对于查询或 MariaDB 服务器状态调优有非常大的帮助。

顾名思义，test 数据库是供测试使用的数据库，其内不含有任何数据表。test 数据库在实际对外提供服务的 MariaDB 服务器中不是必需的，一般都会删除。并且，MariaDB 服务器的权限信息中，任何人都有权限查看含有“test”的数据库。因此，最好在实际运行的 MariaDB 中删除该权限。

```
mariadb> SELECT * FROM mysql.user\G
*************************** 1. row ***************************
                Host: %
                  Db: test\_%
                User:
...

mariadb> SELECT * FROM mysql.db\G
*************************** 1. row ***************************
                Host: %
                  Db: test
                User:
```

```
...
*************************** 2. row ***************************
                Host: %
                  Db: test\_%
                User:

mariadb> DELETE FROM mysql.user WHERE USER='' AND host='%';
mariadb> DELETE FROM mysql.db WHERE USER='' AND host='%' AND db LIKE 'test%';
mariadb> FLUSH PRIVILEGES;
```

如果不删除 test 数据库的不必要的权限，那么任何人都能访问名称以“test”开头的数据库。比如有名为“test_project”的数据库，MariaDB 服务器中的所有用户都能查看该数据库的数据。

3.3.4 新建数据库

使用 CREATE DATABASE 命令可以创建新的数据库。新建数据库时，可以同时为数据库中的数据表指定要使用的字符集与排序方式。

```
mariadb> CREATE DATABASE firstdb;
mariadb> CREATE DATABASE firstdb DEFAULT CHARACTER SET utf8mb4;
```

上述第一个示例中，创建数据库时只明确指定了数据库的名称，而没有指定要用的字符集与排序方式，这样创建出的数据库将采用 MariaDB 服务器设置的默认字符集及该字符集默认的排序方式。若想在创建数据库时为其指定其他字符集，可以在 CREATE DATABASE 命令后加上 DEFAULT CHARACTER SET 子句，明确指定要用的字符集。

使用 DROP DATABASE 命令可以将服务器中已经存在的数据库删除。如果要删除的数据库中含有大量的数据表与记录，那么删除数据库时将会耗费很长的时间，从实际运行的 MariaDB 服务中删除数据库时请注意这点。

```
mariadb> DROP DATABASE firstdb;
```

3.3.5 创建用户

MariaDB 10.0 之前的版本中，创建用户后，还要根据用户账号类型设置不同权限。但是从 MariaDB 10.0.5 开始，除了可以使用以上方式外，还可以使用权限组为用户设置权限。前面已经讲过权限组相关内容，本节将只讲解传统方法。先创建用户，再为不同用户授予不同权限。

1. 创建用户与授予权限

创建新用户时，可以使用 CREATE USER 或 GRANT 命令。使用 CREATE USER 命令只是创建新用户，实际并未授予权限，所以还要使用 GRANT 命令。使用 GRANT 命令授予用户权限时，若授予权限的用户不存在，则会先创建用户（CREATE USER），再为创建的用户授予指

定的权限。因此，创建新用户并为其授予权限时，可以只使用 GRANT 命令，而不必另外再使用 CREATE USER 命令创建用户。

```
mariadb> GRANT SHOW DATABASES ON *.* TO 'user'@'%' IDENTIFIED BY 'user_password';
mariadb> GRANT SELECT, INSERT, UPDATE, DELETE ON firstdb.* TO 'user'@'%';
mariadb> FLUSH PRIVILEGES;
```

上述示例中，SHOW DATABASES 权限是全局权限，而 SELECT 与 INSERT 等权限是数据库或数据表级别的权限，所以不能只使用一条 GRANT 命令将它们一起授予指定的用户。第一条 GRANT 命令中，使用“*.*”权限限定符授予全局权限。第二条 GRANT 命令中，将 firstdb 数据库所有表的 INSERT、UPDATE、SELECT、DELETE 权限授予用户。

在 MariaDB 服务器中创建好新用户并不意味着用户数据库也已准备好，也就是说，用户与数据库是分开管理的。因此，创建数据库后，还要为其另外创建用户，并为用户授予相应权限，这样用户才能正常访问该数据库。

2. 管理员账号

名为 root 的管理员账号众所周知，这可能会给数据库系统带来风险。为了规避该风险，一般要将 root 更换为其他 ID。本书将使用 admin 这个 ID 作为管理员账户名，从安全角度讲，这也不是一个好的选择，建议选用一个其他人难以猜测的账户名。一般而言，管理员账号拥有所有权限。授予权限时，建议只将所有权限授予 'admin'@'localhost' 与 'admin'@'127.0.0.1'，使管理员只能在安装服务器的机器上连接服务器，这在一定程度上可以提高数据库系统的安全性。

```
mariadb> GRANT ALL ON *.* TO 'admin'@'localhost' identified BY 'adminpass' WITH GRANT OPTION ;
mariadb> GRANT ALL ON *.* TO 'admin'@'127.0.0.1' identified BY 'adminpass' WITH GRANT OPTION ;
mariadb> FLUSH PRIVILEGES;
```

上述两个命令用于创建 admin 管理员账号。默认情况下，执行 GRANT 命令后会立即创建账号。如果没有，使用 FLUSH PRIVILEGES 命令刷新即可。

3.4 创建与修改数据表 MariaDB MySQL

在 MariaDB 服务器中创建、删除、修改数据表等操作几乎都使用标准 ANSI SQL 语句，但明确指定存储引擎或记录格式时，MariaDB 服务器有一些略微不同的选项。当然，不用这些选项也可以使用默认值创建数据表，但使用它们可以获得更好的性能。使用不同存储引擎的数据表中，可以将这些选项包含在一条 SQL 语句，以进行连接或子查询等操作。

在 MariaDB 服务器中创建各个数据表时，可以分别为其指定不同存储引擎。决定一个数据表应该使用哪种存储引擎是相当困难的，因为选用的存储引擎会对性能产生很大影响。对于数据表应该采用哪种存储引擎、使用哪种格式，后面讲解 MariaDB 支持的存储引擎时会进行详细说明，此处只学习创建、删除数据表的方法。除此之外，还要学习从 MariaDB 10.0 开始支持的

在线修改 Schema 功能。

3.4.1 创建数据表

MariaDB 服务器中，使用 CREATE TABLE 命令即可创建数据表。略微特别的是，该命令还同时包含 IF NOT EXISTS 条件，如下所示。

```
mariadb> CREATE TABLE [IF NOT EXISTS] tab_test (
            tid BIGINT NOT NULL AUTO_INCREMENT,
            tname VARCHAR(100) NOT NULL,
            tmemo TEXT NOT NULL,
            PRIMARY KEY (tid),
            INDEX ix_tname_tid (tname, tid)
         ) ENGINE=InnoDB;
```

创建数据表或列时，通常使用 IF NOT EXISTS 条件。要创建的表或列已经存在而尝试再次创建时，使用 IF NOT EXISTS 条件将不会产生错误，只会产生警告信息。很难准确判断某个数据表是否已经存在时，通常使用 IF NOT EXISTS 条件语句。

数据表创建语句的最后部分使用了 ENGINE 关键字，表示数据表要采用的存储引擎。上述示例创建 tab_test 数据表时，通过 ENGINE 关键字同时为数据表指定了要使用的存储引擎为 InnoDB。创建数据表时，若未同时为其明确指定要使用的存储引擎，MariaDB 服务器会将设置文件（my.cnf 或 my.ini）中的默认存储引擎进行指定。此处只要记住，使用 ENGINE 关键字可以为数据表指定要使用的存储引擎。

创建数据表后，使用 SHOW CREATE TABLE 命令或 DESC（DESCRIBE）命令可以查看所建数据表的详细内容。

```
mariadb> SHOW CREATE TABLE tab_test;
+----------+-------------------------------------------------------+
| tab_test | CREATE TABLE 'tab_test' (
  'tid' bigint(20) NOT NULL AUTO_INCREMENT,
  'tname' varchar(100) NOT NULL,
  'tmemo' text NOT NULL,
  PRIMARY KEY ('tid')
  KEY 'ix_tname_tid' ('tname','tid')
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
+----------+-------------------------------------------------------+

mariadb> DESC tab_test;
+-------+--------------+------+-----+---------+----------------+
| Field | Type         | Null | Key | Default | Extra          |
+-------+--------------+------+-----+---------+----------------+
| tid   | bigint(20)   | NO   | PRI | NULL    | auto_increment |
```

```
| tname | varchar(100) | NO   | MUL | NULL    |                |
| tmemo | text         | NO   |     | NULL    |                |
+-------+--------------+------+-----+---------+----------------+
```

SHOW CREATE TABLE 命令显示使用 CREATE TABLE 命令创建数据表的过程，DESC 命令以表格形式显示数据表列、索引信息。虽然使用 DESC 命令能够将数据表的组成结构清晰地显示出来，但是使用 SHOW CREATE TABLE 命令直接显示 CREATE 语句更有利于快速把握数据表的组成结构。因为在执行 DESC 命令返回的结果中，很难快速掌握多个数据列形成的复合索引。

3.4.2 修改数据表（在线与离线）

MariaDB 10.0 与 MySQL 5.6 开始引入了在线修改 Schema 的功能。下面对 MariaDB 5.5 与之前版本中使用的离线修改 Schema、从 MariaDB 10.0 开始支持的在线修改 Schema 功能进行讲解。

1. 离线修改 Schema

使用 ALTER TABLE 命令可以向已有数据表插入或删除数据列，也可以添加或删除索引。当然，添加或删除索引时也可以使用 CREATE INDEX 或 DROP INDEX 命令。但使用 ALTER TABLE 命令可以对其进行集中处理，所以使用 ALTER TABLE ADD INDEX 与 ALTER TABLE DROP INDEX 命令要比使用 CREATE INDEX 与 DROP INDEX 命令更好。

下列示例先向 tab_test 数据表添加 created 数据列，然后在之上创建索引。如示例所示，执行修改 Schema 语句后，后面会显示处理了几条记录、出现了几个重复记录，还有其他警告信息。执行完重要的处理后，要注意在输出信息中查看重复与警告信息，以确认执行结果。

```
mariadb> ALTER TABLE tab_test ADD created DATETIME NOT NULL;
Query OK, 0 rows affected (0.57 sec)
Records: 0  Duplicates: 0  Warnings: 0

mariadb> ALTER TABLE tab_test ADD INDEX ix_created (created);
Query OK, 0 rows affected (0.42 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

MariaDB 5.5 与 MySQL 5.5 之前的版本不支持在线修改 Schema 功能。在这些版本的服务器中修改 Schema 时，一般先将记录复制到临时表，然后删除原表，将临时表改为原表名。具体修改步骤如下。

```
(1) 锁定 tab_test 数据表;
(2) 创建包含请求修改内容的临时表;
(3) 从 tab_test 数据表逐条读出记录，复制到步骤 2 创建的临时表;
(4) 反复执行步骤 3，直到 tab_test 数据表中的所有记录全部复制到临时表;
(5) 删除 tab_test 数据表，将临时数据表重命名为 tab_test;
(6) 解除 tab_test 锁定。
```

修改数据表的结构时，其他连接无法修改该数据表的记录。因此，修改拥有大量记录数据表的 Schema 时，不能在正在提供服务的数据库中进行，要在服务处于停止状态时修改 Schema。分别使用不同语句为 tab_test 数据表创建列与索引时，由于要对 tab_test 数据表中的记录复制两次，所以修改 Schema 的时间变为原来的两倍。因此，MariaDB 中，一般将多个修改 Schema 的语句放在一条 ALTER TABLE 语句中执行，如下所示。

```
mariadb> ALTER TABLE tab_test ADD created DATETIME NOT NULL,
                              ADD INDEX ix_created (created);
Query OK, 0 rows affected (0.88 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

当然，MariaDB 5.5 之前的版本中，并非所有对 Schema 的修改都以复制记录的形式进行。下面列出了一些典型的修改 Schema 的情形，对它们进行修改处理时，并不需要将数据表中的所有记录复制到临时数据表。

- 修改列名。
- 修改数值类型表示的长度（由 INT(2) 变为 INT(3)）。
- 修改数据表的注释。
- 向 ENUM 类型项目列表最后添加新项目。
- 修改数据表名。

2. 在线修改 Schema MariaDB

在 MySQL 5.5 用户手册中查看 ALTER TABLE 语句的语法时，可以看到有一个 ONLINE 关键字可用。但 MySQL 5.5 版本只是根据 ANSI SQL 标准添加了 ONLINE 关键字，实际上并未使用。当然，在 MySQL Cluster 中可以使用 ONLINE 关键字，但本书内容将不涉及 MySQL Cluster。在 MySQL 5.5 中使用 ALTER ONLINE TABLE...语句会产生错误。MariaDB 从 5.3 版本开始提供对 ONLINE 关键字的支持，使用该关键字可以确定处理中是否实际需要复制记录。

在 MariaDB 5.5 中进行如下测试，可以轻松掌握 ONLINE 关键字在 MariaDB 中是如何使用的。

```
mariadb> CREATE TABLE t1 (fd1 INT, fd2 ENUM('red','green'));
mariadb> ALTER ONLINE TABLE t1 MODIFY fd2 ENUM('red','green','blue');
-> Query OK, 0 rows affected (0.11 sec)
-> Records: 0  Duplicates: 0  Warnings: 0

mariadb> ALTER ONLINE TABLE t1 ADD fd3 INT;
-> ERROR 1656 (HY000): Can't execute the given 'ALTER' command as online
```

上述示例中，第一条命令创建了名为 t1 的数据表，第二条命令修改 fd2 数据列的 ENUM 类型包含的项目，在 ENUM 中新增一个 blue 项。前一小节已经讲过，在 ENUM 的项列表末尾添加新项时，可以不复制数据表记录而只修改结构。因此，示例中的第二条命令即使包含

ONLINE 关键字也仍然能够正常执行，不会出现问题。但是第三个 ALTER 命令中使用了 ONLINE 关键字，执行后显示错误信息，表明无法执行。

MariaDB 将 ALTER 命令中的 ONLINE 选项用作安全装置，使得仅在不需要复制记录的情形下可以修改 Schema。在 MariaDB 中要修改 Schema 时，若不明确是否需要复制数据表中的所有记录，那么只要在要执行的命令中添加上 ONLINE 关键字即可。

下面介绍 MariaDB 10.0 版本开始新增的在线修改 Schema 功能。所谓的在线修改 Schema 是指，一个连接正在修改数据表结构时，其他连接的 INSERT、UPDATE、DELETE 语句可以马上执行处理而不必进入等待状态。

从 MariaDB 10.0 开始，典型的数据表 Schema 修改命令都可以在线进行处理，比如添加或删除数据列、添加或删除索引。目前支持在线修改 Schema 的存储引擎有 InnoDB、TokuDB，当前发布的 MariaDB 10.0.8 中，支持在线处理的 DDL 命令有：

- 添加与删除数据列
- 创建与删除索引
- 更改数据列顺序

原有用于增加数据列长度的在线修改 Schema 的命令虽然包含于 MariaDB 10.0 版本的改善计划，但 MariaDB 10.0.8 Beta 版本并不能使用在线修改功能。

```
mariadb> ALTER ONLINE TABLE tab_alter MODIFY fd VARCHAR(20);
ERROR 1846 (0A000): LOCK=NONE is not supported. Reason: Cannot change column type INPLACE.
Try LOCK=SHARED.
```

上述示例中，数据列 fd 创建时为 VARCHAR(10)，使用 ALTER ONLINE 命令将其改为 VARCHAR(20) 时，就会产生错误并显示错误信息："Reason: Cannot change column type INPLACE"，详细告知在线修改 Schema 失败的原因。错误信息中的"INPLACE"表示不复制数据表记录并修改 Schema，错误信息表明，无法在不复制记录的情形下修改 Schema。

MariaDB 10.0 版本的改善计划中包含如下四种处理情形，但 MariaDB 10.0.8 版本尚未实现。

- 增加 CHAR 或 VARCHAR 数据列的长度（e.g. 将 VARCHAR(10) 更改为 VARCHAR(20)）。
- 增加数值类型数据列的存储空间（e.g. 将 TINYINT 更改为 INT）。
- OPTIMIZE 命令。
- ANALYZE 命令。

此外，MariaDB 10.0 改善计划已经包含下面几种 Schema 修改，但尚未支持在线修改 Schema。

- 减小数据列长。
- 修改数据列名。
- 修改字符串数据列的字符集与排序方式。
- 添加 AUTO_INCREMENT 数据列。
- 添加与删除主键。

MariaDB 10.0.8 中的 InnoDB 与 XtraDB 存储引擎都由 MySQL 5.6 的 InnoDB 派生而来，所以在 MariaDB 中在线修改 Schema 与 MySQL 5.6 的行为方式几乎是一样的。MariaDB 的 XtraDB 存储引擎是 MySQL 5.6 InnoDB 存储引擎的改进版本，所以它比 Oracle MySQL 5.6 的 InnoDB 存储引擎拥有更多的功能，伸缩性更好。因为 XtraDB 存储引擎是 Percona 公司在 MySQL 5.6 InnoDB 存储引擎源码的基础上经过改善开发推出的。

正如在 MariaDB 介绍中提及的，MariaDB 吸收了 MySQL 的大部分功能，MariaDB 修改 Schema 功能与 MySQL 的在线修改 Schema 功能没什么不同。因此，接下来将讨论 MySQL 5.6 的在线修改 Schema 功能。MySQL 的在线修改 Schema 功能有 LOCK 与 ALGORITHM 选项，它们也可以在 MariaDB 中使用，请务必牢记。

3. MySQL 的在线修改 Schema 功能 MariaDB MySQL

对于 MySQL 5.6 的 InnoDB 数据表，在线修改 Schema 时并不需要使用什么特别的 ALTER TABLE 命令或选项，与 MySQL 5.5 中使用的命令或选项是一样的。但有时所用的 ALTER TABLE 语句的语法略微有些不同，有时还需要对系统进行设置。此时，在 ALTER TABLE 语句中明确使用 LOCK 与 ALGORITHM 子句即可控制在线修改 Schema 的处理方法。对数据表进行并发处理时，LOCK 子句为其提供了调优功能，而 ALGORITHM 子句提供了与在线 DDL 性能相关的调优功能。

LOCK={DEFAULT|NONE|SHARED|EXCLUSIVE}

在线执行 DDL 时，若允许其他连接对数据表进行读写，则设置 LOCK=NONE；若只允许读取，则设置 LOCK=SHARED。但 DDL 语句无法满足给定的 LOCK 选项时，就会引发错误并立即终止。设置 LOCK=DEFAULT 相当于不明确使用 LOCK 子句。

ALGORITHM={DEFAULT|INPLACE|COPY}

在线修改 Schema 算法中，COPY 是 MySQL 5.5 之前版本使用的方法。使用该方法将数据表中的所有记录复制到临时数据表，复制完成后，再将临时表重命名为原表。而 INPLACE 是从 MySQL 5.6 开始支持的修改 Schema 的方法，使用时并不需要将数据表中的记录复制到临时表，而是在原位置（inplace）直接进行修改。设置 ALGORITHM=DEFAULT 相当于不明确使用 ALGORITHM 子句。将 INPLACE 算法应用于在线修改 Schema 时，MySQL 服务器会按如下步骤进行操作：

(1) 检查数据表所用存储引擎是否支持以 INPLACE 方式修改 Schema；
(2) 为以 INPLACE 方式修改 Schema 作准备：准备关于新建索引的信息，并为在线修改 Schema 期间追踪修改数据作准备；
(3) 修改数据表 Schema 及记录新 DML：实际修改 Schema，在此期间，其他连接的 DML 操作不会阻塞。像这样，在线修改 Schema 期间，其他线程中用户发出的 DML 命令会记录到另外的日志；
(4) 执行日志记录：在线修改 Schema 期间，其他用户发出的 DML 命令会保存到另外的日志，该步骤将对步骤 3 新构建的数据表执行这些 DML 命令；
(5) 以 INPLACE 方式完成（COMMIT）Schema 修改。

采用 INPLACE 算法修改 Schema 时，步骤 2 与步骤 4 暂时需要排他锁（Exclusive lock）。此时，其他连接的 DML 将会短暂等待。实际的修改操作在步骤 3 中进行，耗费的时间也比较多。但该过程中，其他连接的 DML 操作不会阻塞。采用 INPLACE 算法在线修改数据表期间，新到来的 DML 查询修改的信息会放入名为“在线修改日志”（Online alter log）的内存空间。在线修改 Schema 完成后，记录在日志中的内容会应用到新表。请注意，“在线修改日志”只在内存中（非磁盘中）创建，它的大小由 innodb_online_alter_log_max_size 系统变量确定。“在线修改日志”默认大小为 128 MB，在线修改 Schema 要耗费较长时间或修改期间会有大量 DML 查询到来时，建议各位将“在线修改日志”的大小设置得更大一些。innodb_online_alter_log_max_size 系统变量是一个基于会话级别的动态变量，需要时可以随时修改。

注意 我没有在实际服务环境的 MySQL 服务器中使用在线修改 Schema 功能的经验，但在各种测试中遇到过几次“在线修改日志”不够用的情形。测试环境中，DML 查询很多，数据表的记录也有很多，在线修改 Schema 要花费很长时间，此时就有可能出现“在线修改日志”不够用的问题。还有一个问题是，我们很难使用在线 Schema 修改工具预测“在线修改日志”到底需要设置为多大才合适。而且在线修改 Schema 过程中也无法获知“在线修改日志”内存空间使用了多少、剩余多少。

有一个变通的办法是，对到来的 DML 查询数以及各条 DML 查询引起修改的字节数大致进行预测。如果可以计算每个数据列或记录所需空间开销当然更好，但即使无法计算，也可以将数据列的数据大小与记录数相乘，再与当前 innodb_online_alter_log_max_size 的值进行比较，之后再进行在线修改 Schema 操作。

一大早就起床，花了好几个小时进行修改，最后因 innodb_online_alter_log_max_size 设置不够而撤销，这该是一件多么让人痛心疾首的事啊！

处理 ALTER TABLE 命令时，若想和 MySQL 5.5 之前版本那样将全部记录复制到临时表，可以将 old_alter_table 系统变量设置为 ON，或者在 ALTER TABLE 命令中使用 ALGORITHM=COPY 选项。若 old_alter_table 系统变量与 ALGORITHM 设置值冲突，则优先使用 ALGORITHM 的设置值。ALGORITHM=DEFAULT 表示不使用任何算法。

注意 请注意，MySQL 5.6 或 MariaDB 10.0 中，在线修改 Schema 时，启用 ALGORITHM=INPLACE 选项并不意味着处理过程中不会复制记录。向数据表进行添加或删除数据列等操作时，虽然采用了 INPLACE 算法进行处理，但是实际磁盘上的数据文件还是进行了复制。因此，INPLACE 算法不能用于判断数据表中的记录是否复制，但可以判断修改 Schema 期间，其他连接中有 DML 到来时是否允许处理这些查询。下列示例是采用 ALGORITHM= INPLACE 方式添加数据列的过程中对数据目录内容进行查询的结果，从中可以看到，期间创建了临时表并复制了记录。

```
shell> ls -alh /mariadb_data_dir/
-rw-rw---- 1 mysql dba 8.5K 12 月 26 12:25 #sql-5e11_5.frm
-rw-rw---- 1 mysql dba 228M 12 月 26 12:25 #sql-ib842.ibd
-rw-rw---- 1 mysql dba 8.5K 12 月 26 12:24 tab_test.frm
-rw-rw---- 1 mysql dba 232M 12 月 26 12:25 tab_test.ibd
...
```

因此，MySQL 5.6用户手册将采用INPLACE算法进行处理但又复制数据表记录的形式称为“REOGRANIZE”，将不能采用INPLACE算法进行处理并采用复制数据表记录的形式称为“REBUILD”。查看用户手册时，请注意这两个词的区别。

表3-1表示在MySQL 5.6中，在线修改Schema时是否要使用IN-PALCE算法，以及其他连接的DML（INSERT、UPDATE、DELETE）是否阻塞。

表3-1 在线修改Schema（Online DDL）的各种处理方式

修改内容	IN-PLACE	执行其他连接的DML	执行其他连接的SELECT	说明
添加索引	O	O	O	不包括创建FULLTEXT INDEX（参考CREATE FULLTEXT INDEX说明）
添加全文索引	O	X	X	创建第一个FULLTEXT INDEX时需要复制记录
删除索引	O	O	O	
设置数据列的DEFAULT值	O	O	O	仅修改*.FRM文件，完全不修改实际数据文件
修改AUTO_INCREMENT值	O	O	O	仅修改MySQL服务器内存的AutoIncrement值，不修改磁盘上的数据文件或FRM文件
创建外键	O	O	O	为防止复制数据表中的记录，创建FOREIGN KEY时，需要禁用foreign_key_checks系统变量
删除外键	O	O	O	
更改列名	O	O	O	修改Schema时，若允许处理其他连接的DML，则需要让数据列的类型与其他信息全部相同，保证仅有列名不同
添加数据列	O	O	O	添加AUTO_INCREMENT数据列时，禁止执行其他连接的DML。可以使用IN-PLACE算法，但实际上内部仍需要复制数据表记录，系统负荷高
删除数据列	O	O	O	允许使用IN-PLACE算法，但实际上内部仍需要复制数据表记录，系统负荷高
修改列序	O	O	O	
修改数据表的ROW_FORMAT	O	O	O	
修改数据表的KEY_BLOCK_SIZE	O	O	O	
修改数据列为NULLABLE	O	O	O	
修改数据列为NOT NULL	O	O	O	
修改数据列类型	X	X	O	禁止使用IN-PLACE算法，禁止执行其他连接中的DML

（续）

修改内容	IN-PLACE	执行其他连接的 DML	执行其他连接的 SELECT	说明
添加主键	O	O	O	允许使用 IN-PLACE 算法，但实际上内部仍需要复制数据表记录，系统负荷高。此外，将 NULLABLE 数据列更改为 NOT NULL 并添加主键时，不可使用 IN-PLACE 算法，此时也不可执行其他连接的 DML
删除主键 +（同时）添加主键	O	O	O	此情形为 ALTER TABLE 语句同时包含 DROP PRIMARY KEY 与 ADD PRIMARY KEY。允许使用 IN-PLACE 算法，但实际上内部仍需要复制数据表记录，系统负荷高
删除主键	X	X	O	删除原有主键，但不添加新主键
修改或定义数据列字符集	X	X	O	需要重建数据表
与 FORCE 选项一起执行 REBUILD	X	X	O	需要重建数据表。与将 ALGORITHM 设置为 COPY 或将 alter_table 系统变量设置为 1 一样

在线修改 Schema 语句中，即使使用 IN-PLACE 算法，有时也会有复制数据表记录这一过程。事实上，除了添加或删除索引之外，其他所有操作几乎都伴随着创建临时表并将数据表中的所有记录复制到临时表的过程。

无论是 MySQL 5.6 还是 MariaDB 10.0，LOCK 与 ALGORITHM 选项与 ONLINE 关键字（讲解在线修改 Schema 时提到过）所起的作用几乎都是一样的。下列示例向 tab_test 数据表添加新的数据列时，同时使用了 LOCK=NONE 与 ALGORITHM=INPLACE 选项。

```
mariadb> ALTER TABLE tab_test ADD fd2 VARCHAR(20),
                                  LOCK=NONE, ALGORITHM=INPLACE;

mariadb> ALTER TABLE tab_test ADD fd0 INT AUTO_INCREMENT PRIMARY KEY,
                                  LOCK=NONE, ALGORITHM=INPLACE;
```

正如表 3-1 所示，上述两个查询中，第一个查询会正常运行而不会出现任何问题；但第二个查询中，向数据表添加 AUTO_INCREMENT 数据列并在其上创建了 PRIMARY KEY，此时将无法使用 INPLACE 算法，即无法进行在线处理。下面是以上两个查询在 MariaDB 10.0 中的执行结果。与预料中的一样，第一个查询得到在线处理，没有出现任何问题；但执行第二个查询时，由于添加 AutoIncrement 时需要加锁，所以产生错误，显示出错信息，要求将 LOCK=NONE 修改为 LOCK=SHARED（Reason: Adding an auto-increment column requires a lock. Try LOCK=SHARED）并终止。

```
mariadb> ALTER TABLE tab_test ADD fd2 VARCHAR(20), LOCK=NONE, ALGORITHM=INPLACE;
Query OK, 0 rows affected (0.24 sec)
```

```
Records: 0  Duplicates: 0  Warnings: 0

mariadb> ALTER TABLE tab_test ADD fd0 INT AUTO_INCREMENT PRIMARY KEY,
                              LOCK=NONE, ALGORITHM=INPLACE;
ERROR 1846 (0A000): LOCK=NONE is not supported. Reason: Adding an auto-increment column
requires a lock. Try LOCK=SHARED.
```

像这样，如果使用 LOCK=NONE 与 ALGORITHM=INPLACE 选项能够像 MariaDB 的 ONLINE 关键字一样可以在线修改 Schema，则立即执行，否则终止。这相当于为向外提供服务的 MariaDB 加了一层安全网。

在 MariaDB 10.0 或 MySQL 5.6 中在线修改 Schema 时，可能有以下几种原因导致失败。

(1) 长时间执行 ALTER TABLE 命令的同时，其他连接大量执行 DML；或者在线修改日志空间不足时，在线修改 Schema 都会失败。

```
ERROR 1799 (HY000) : Creating index 'idx_aaa' required more than
                     'innodb_online_alter_log_max_size' bytes of modification log.
                     Please try again.
```

(2) 执行 ALTER TABLE 命令期间，ALTER TABLE 之前版本的表结构中不会有任何问题。但向 ALTER TABLE 之后的表结构插入（INSERT）或更新（UPDATE）不适合的记录后，将导致在线修改 Schema 失败。

```
ERROR 1062 (23000): Duplicate entry 'd005-10001' for key 'PRIMARY'
```

(3) 修改 Schema 时使用了较低级别的锁选项。

```
ERROR 1846 (0A000): LOCK=NONE is not supported. Reason: Adding an auto-increment
                    column requires a lock. Try LOCK=SHARED.
```

(4) 使用 LOCK=NONE 选项在线修改 Schema 时，修改的初始与最终阶段需要加锁，但因无法得到锁而导致超时，最终失败。

```
ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction
```

(5) 在线创建索引过程中，使用 tmpdir 系统变量中设置的磁盘临时目录进行排序时，目录空间不足将导致在线修改 Schema 失败。

注意 启用 LOCK=NONE 选项在线修改 Schema 时，修改的初始与最终阶段需要对数据表的元数据进行锁定。该过程中若无法锁定数据表的元数据，将会导致在线修改 Schema 因超时而失败。MySQL 服务器或 MariaDB 中存在大量关于超时的系统变量，用作超时浏览的系统变量是哪个呢?

若数据表采用 InnoDB 存储引擎，在线修改 Schema 时需要用数据表级别的元数据锁。MariaDB 与 MySQL 中，元数据锁的超时设定由 lock_wait_timeout 系统变量确定。请注意，innodb_lock_wait_timeout 用于设置 InnoDB 记录锁超时时间，它与在线修改 Schema 无关。下面使用 SHOW GLOBAL VARIABLES 命令查看 lock_wait_timeout 变量的设置值。

```
mariadb> SHOW GLOBAL VARIABLES LIKE 'lock_wait_timeout';
+----------------------------+----------+
| Variable_name              | Value    |
+----------------------------+----------+
| lock_wait_timeout          | 31536000 |
+----------------------------+----------+
```

从返回的结果可以看到，lock_wait_timeout 设置为 31 536 000 秒，该时间是 MySQL 与 MariaDB 服务器的默认设置值。在线修改 Schema 时，有其他连接长时间执行 DML，或者因事务非正常终止而仍有连接的插入（INSERT）、更新（UPADTE）查询处于事务激活状态时，31 536 000 秒时间内，在线修改 Schema 操作会一直等待。

在线修改 Schema 时要注意，不要使妨碍修改 Schema 的事务出现。但如果无法避免这种情况，则可以将 lock_wait_timeout 设置为一个合适的时间。超过指定的等待时间后，在线修改 Schema 操作将取消。lock_wait_timeout 是全局变量，也是会话变量，所以一定要在执行在线修改 Schema 的会话中将 lock_wait_timeout 设置为合适的值。

```
mariadb> SET SESSION lock_wait_timeout=1800;
mariadb> ALTER TABLE tab_test ADD fd2 VARCHAR(20), LOCK=NONE, ALGORITHM=INPLACE;
```

4. pt-online-schema-change

pt-online-schema-change 是在线修改 Schema 的第三方工具，包含于 Percona 工具包（http://www.percon.com/doc/percona-toolkit/2.2/）。pt-online-schema-change 虽然由 Percona 公司开发，但是无论在 MySQL 还是 MariaDB 服务器中都能使用。该款工具应用广泛，相关网站也很多，详细内容不再赘述，下面只讲解其工作原理及几个需要注意的事项。

使用 pt-online-schema-change 工具修改 Schema 时，会先创建一个与原数据表（该表为待修 Schema 的数据表）拥有相同结构的新表，然后将原表中的数据逐步复制到新表。比如有一个拥有 emp_no、first_name、last_name、hire_date 数据列的 employees 表，向该表添加名为 birth_date 的新列时，要使用如下命令。

```
shell> pt-online-schema-change --alter \
  "ADD birth_date DATE" D=employees,t=employees \
  --no-drop-old-table \
  --no-drop-new-table \
  --chunk-size=500 \
  --chunk-size-limit=600 \
  --defaults-file=/etc/my.cnf \
  --host=127.0.0.1 \
  --port=3306 \
  --charset=UTF8MB4 \
  --user=dba_user \
  --ask-pass \
```

```
--progress=time,30 \
--max-load="Threads_running=100" \
--critical-load="Threads_running=1000" \
--chunk-index=PRIMARY \
--retries=20 \
--execute
```

pt-online-schema-change处理上述命令时，会先根据原employees数据表的结构创建新表（_employees_new），并将要添加的列加到新表，如图3-2所示。然后创建AFTER触发器，使对原数据表（employees）的INSERT、UPDATE、DELETE操作内容能传递到_employ_new数据表。之后，pt-online-schema-change读取原数据表中的记录，所读条数由chunk-size选项确定，再将读取的记录复制到_employees_new数据表。pt-online-schema-change以内存块（Chunk）为单位复制记录时，用户对原employees数据表执行的DML都会通过触发器自动反映到_employees_new数据表。pt-online-schema-change复制完所有记录后，使用RENAME命令将employees表更名为_employees_old，将_employees_new表名修改为emplyees。RENAME命令会将多个数据表的更名操作作为一个事务进行处理，所以更名过程中，用户的查询不会失败。这样更名完成后，应用程序就可访问添加了新数据列的数据表，就像什么事也没发生一样。让人略感意外的是，实现在线修改Schema的方法竟然如此简单。

使用pt-online-schema-change工具时，要着重注意如下两个方面。

(1) 触发器重复

如前所述，使用pt-online-schema-change工具对数据表进行INSERT、UPDATE、DELETE操作时，要在这些语句中创建AFTER触发器。如果原数据表中已经存在AFTER触发器，那么就无法再使用pt-online-schema-change工具了。

(2) 死锁（Dead Lock）

另一个要注意的是死锁问题。假设employees数据表的first_name列有唯一索引（Unique index）。pt-online-schema-change工具会使用主键，以内存块为单位分割记录，然后将所选范围内的记录复制到_employees_new数据表。这一过程中，对_employees_new数据表的主键加排他锁后，为了进行重复检查，还要对first_name数据列的唯一索引加锁。此外，用户修改的内容（INSERT、UPDATE、DELETE）也通过触发器应用到_employees_new数据表。但是，通过触发器到来的查询会先对first_name数据列中创建的唯一索引（非主键）加排他锁，然后试图再次向记录的主键加锁时，就会与pt-online-schema-change进程的以内存块为单位进行的复制作业产生死锁。

像这样，在拥有主键与唯一键（Unique key）的数据表中，死锁现象是很常见的。如果主键还是AutoIncrement，那就更要注意了。一个更严重的问题是，发生死锁时，InnoDB/XtraDB存储引擎会从两个事务中强制终止记录修改较少的那个。这样，相比于每次修改5~600条记录的

pt-online-schema-change，一个修改 1~2 条记录的用户事务肯定会被强制终止。

pt-online-schema-change 实用工具也支持 --max-load、--critical-load 选项，它们自动调整负荷量。但是 pt-online-schema-change 运行时，还是会对执行应用程序的查询请求产生负面影响。若可以允许 pt-online-schema-change 修改 Schema 的速度略慢一些，则可以对 pt-online-schema-change 实用工具的代码进行修改，使之每次完成 5~600 条记录的 Chunk 复制后，就等待 2~30 毫秒，这对提升系统的稳定性非常有帮助。代码修改如下。

```
...
package pt_online_schema_change;

use strict;
use warnings FATAL => 'all';
use English qw(-no_match_vars);
use constant PTDEBUG => $ENV{PTDEBUG} || 0;

use Time::HiRes qw(time sleep usleep);
use Data::Dumper;
$Data::Dumper::Indent = 1;
$Data::Dumper::Sortkeys = 1;
$Data::Dumper::Quotekeys = 0;

...
...

sub exec_nibble {
  ## Sleep micro seconds : 20000 micro seconds(20 milli-seconds)
  usleep(20000);

  my (%args) = @_;
  my @required_args = qw(Cxn tbl NibbleIterator Retry Quoter OptionParser);
  ...
```

如以上代码所示，修改代码时一定要先通过“use Time::HiRes qw(time sleep usleep)”语句导入 usleep 函数，然后在下面调用 usleep(20000); 函数。

5. 能否使用在线修改 Schema

从 MySQL 历史上看，MariaDB 10.0 与 MySQL 5.6 的在线修改 Schema 功能是人们很久以前就要求增加的，许多人对它满怀期待。那么，在实际向外提供服务的 MariaDB 服务器中，真的要使用在线修改 Schema 功能吗？下面让我们一起进行分析。虽然此处只讨论 MariaDB，但由于 MariaDB 的 InnoDB 与 MySQL 5.6 的 InnoDB 存储引擎一样，所以 MySQL 5.6 中也存在着同样的疑问。

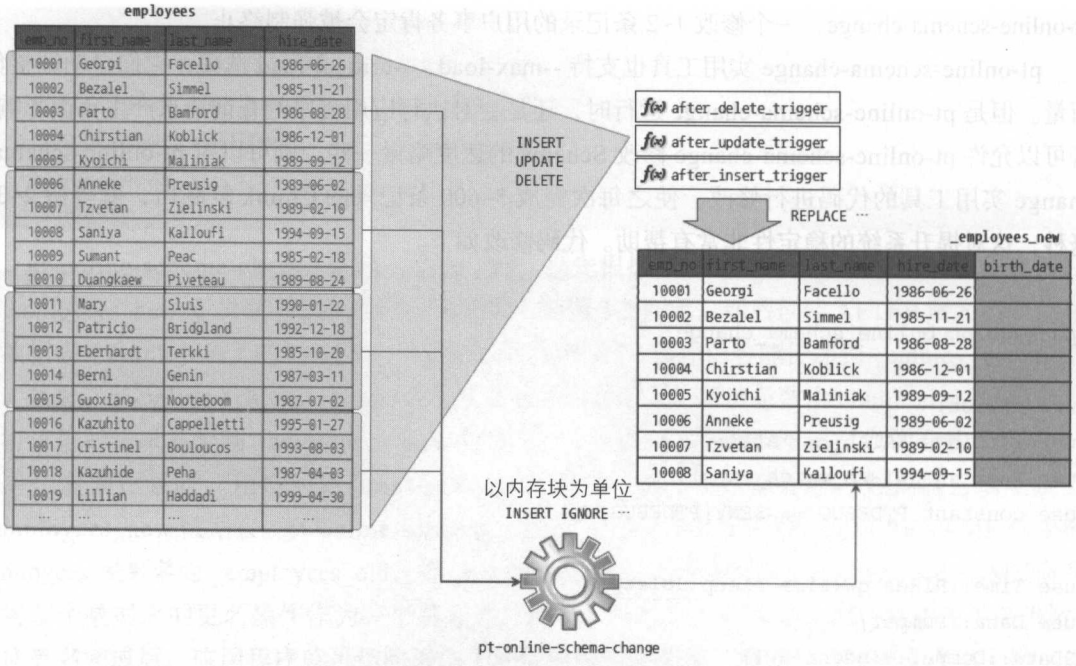


图 3-2 pt-online-schema-change 工作原理

首先构建一个服务环境，要在这个环境中对拥有 1000 万条记录的数据表进行每秒 1000 次的 SELECT 查询，还要每秒执行 270 次左右的 DML 语句（INSERT、UPDATE、DELETE）。并且在数据库中进行在线修改 Schema，收集性能信息，绘制性能图表，如图 3-3 所示。

从图 3-3 中可以看到，在线修改 Schema 大约花了 2 分 40 秒，并且在处理过程中，SELECT 查询与 DML 查询的处理量上下浮动大，表现相当不稳定。原本寄予厚望的在线修改 Schema 功能在运行时显得非常不稳定。

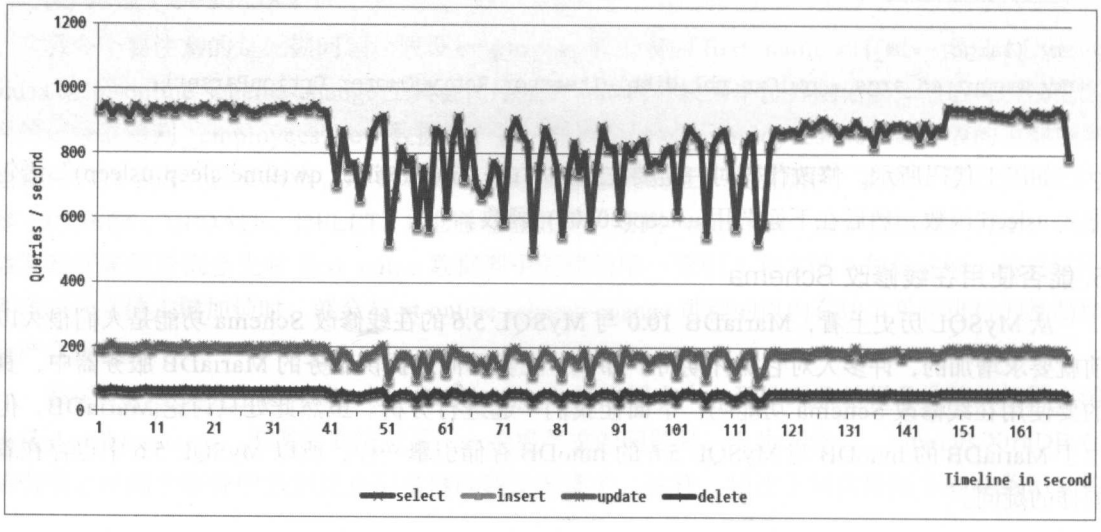


图 3-3 在线修改 Schema

MariaDB 与 MySQL 支持在线修改 Schema 功能之前，用户经常使用 pt-online-schema-change 工具进行在线修改。下面使用 pt-online-schema-change 工具重新进行上述测试。从图 3-4 中可以看到，用 pt-online-schema-change 工具跑完测试大约花了 6 分 20 秒。使用 pt-online-schema-change 工具所花费的时间大约是使用 MySQL 在线修改 Schema 功能的两倍多，其实这是因为我对 pt-online-schema-change 工具的源码进行了一些修改。经过修改的 pt-online-schema-change 工具每完成一个 Chunk（约 500 条记录）复制都要暂停 5 毫秒。从图 3-4 中可以看到，pt-online-schema-change 工具工作时，对查询服务处理完全没有不良影响。

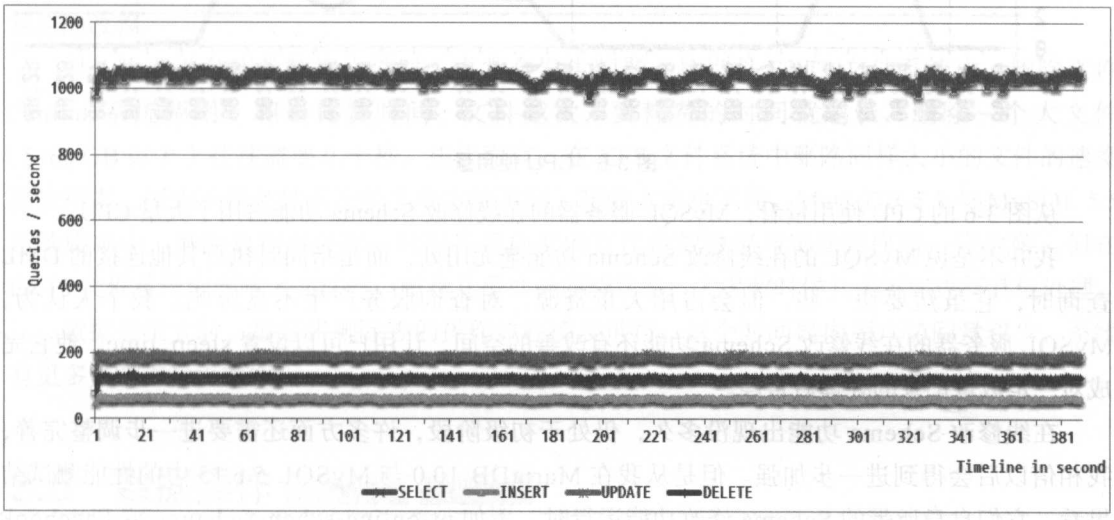


图 3-4 pt-online-schema-change 工具

接下来，对 MySQL 的在线修改 Schema 功能与 pt-online-schema-change 运行期间 CPU 的使用量与系统负载进行测量，绘出图 3-5 与图 3-6。从图 3-5 所示的系统负载图表看，MySQL 的在线修改 Schema 运行时的系统负载约为 3.5，而 pt-online-schema-change 运行时的系统负载大致维持在 1 之内。

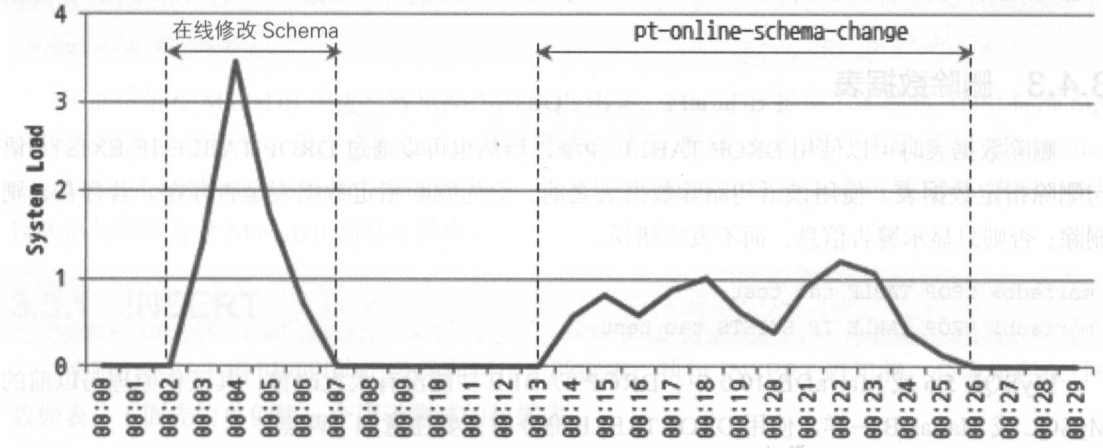


图 3-5 系统负载

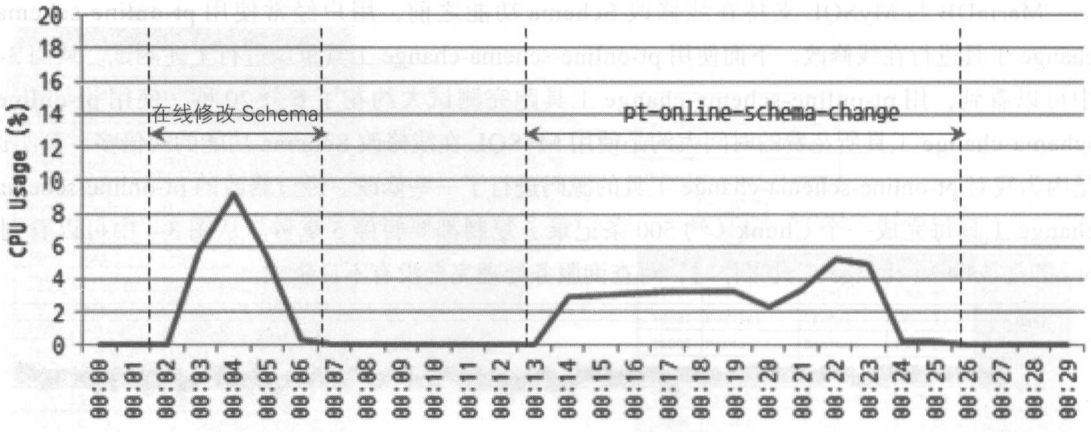


图 3-6 CPU 使用量

从图 3-6 的 CPU 使用量看，MySQL 服务器的在线修改 Schema 功能占用了大量 CPU。

我并不是说 MySQL 的在线修改 Schema 功能毫无用处，而是指同时执行其他连接的 DML 查询时，它虽然要快一些，但会占用大量资源，对查询服务产生不良影响。我个人认为，MySQL 服务器的在线修改 Schema 功能还有改善的空间，让用户可以设置 sleep_time，使它完成对一定数量记录的处理后暂停。

在线修改 Schema 功能出现没多久，仍处于初级阶段，许多方面还需要进一步调整完善，我相信以后会得到进一步加强。但是从我在 MariaDB 10.0 与 MySQL 5.6.15 中的性能测试结果看，它们自身所带的 Schema 修改功能运行时，不如 pt-online-schema-change 或 Facebook 的 OSC 稳定。

注意 pt-online-schema-change 自动创建的触发器中，DELETE 触发器要求要修改的数据表对象有主键或唯一键，否则就无法使用 pt-online-schema-change。MariaDB 或 MySQL 中，要求数据表的分区键（partition key）必须为所有主键与唯一键的一部分。因此，被分区的数据表中，有许多没有主键与唯一键。在这样的数据表中使用 pt-online-schema-change 工具时，需要格外注意。

3.4.3 删除数据表

删除数据表时可以使用 DROP TABLE 命令，当然也可以通过 DROP TABLE IF EXISTS 语句删除指定数据表。使用该语句删除数据表之前，会先判断指定数据表是否存在。若存在，则删除；否则只显示警告信息，而不发生错误。

```
mariadb> DROP TABLE tab_test;
mariadb> DROP TABLE IF EXISTS tab_test;
```

MySQL 5.6 或 MariaDB 10.0 中，DROP TABLE 功能没有太大改善，其工作原理与以前的 MySQL 或 MariaDB 一样。使用 DROP TABLE 命令时，要注意如下两点：

(1) 从缓冲池中删除相关数据表页；

(2) 从物理磁盘中删除数据文件。

InnoDB 缓冲池中可以缓存数据表的数据或索引页（Block），数据表删除后，与其相关的页面也要从缓冲池中删除。为此，执行 DROP TABLE 命令时需要对缓冲池中的所有缓存页面进行扫描。InnoDB 缓冲池很大、使用量很高时，扫描缓冲池中的页面将非常耗费时间。如果先扫描缓冲池，再删除要删除的数据表的缓存块，一般不会出现什么问题。MariaDB 5.5 与 MySQL 5.5 后续版本中，针对于为删除数据表而进行的缓冲池扫描部分，提交了许多 Bug 报告，并进行了相应的改善。

但从物理磁盘删除数据文件时，常常会发生问题。特别是 EXT3 文件系统中，删除文件（unlink 系统调用）相当耗费时间。文件越大，要耗费的时间就越长，删除一个大文件（5~60 GB 以上）往往需要几十秒。相比而言，在 XFS 文件系统中删除同样大小的文件的速度要快得多。因为不同文件系统的实现方法不同，删除方法也不同。MariaDB 5.5 与 MySQL 5.5 后续版本中，删除数据文件时，对服务器内部的互斥量的锁定与解除程序有一定改善。但在 EXT3 文件系统中删除大文件时，磁盘几乎处于崩溃状态，大多数时候几乎不能进行其他处理。

数据表很大时，最好将删除表的操作放在凌晨进行。这个时间段内用户访问量较少，系统有更多资源进行该操作。

3.5 数据操作 MariaDB MySQL

使用 SQL 语句查询或修改数据表中的内容时，一定要明确指出目标数据库名与数据表名。但每次运行 SQL 语句都进行该操作十分麻烦，为了解决这一问题，MariaDB 提供了 USE 命令，使用它可以选定一个数据库，以后进行的所有操作都将只针对该数据库。也就是说，使用 USE 命令选定数据库后，若未在使用各种数据表、存储程序时指定数据库名称，则都会在选定的数据库中查找相关对象（数据表、存储程序等）。

```
mysql>USE 数据库名 ;
```

下面讲解在 MariaDB 中进行数据操作的 SQL 语句。MariaDB 提供了多种与 ANSI 标准 SQL 语句不同的变形语句。关于各 SQL 语句的语法，请参考 MySQL 用户手册页面（http://dev.mysql.com/doc/refman/5.6/en/sql-syntax.htm）。此处只学习 SELECT 查询语句与 DML 语句，使用这些语句可以查看 MariaDB 的基本操作。

3.5.1 INSERT

使用 INSERT 语句可以将记录保存到指定数据表，执行 INSERT 语句前，必须先存在目标数据表。下面先简单创建一个用于测试的数据表。

```
mariadb [(none)]> USE test;
Database changed

mariadb [test]> CREATE TABLE tab_test (
                  fd1 INT NOT NULL,
                  fd2 VARCHAR(50),
                  PRIMARY KEY(fd1)) ENGINE = InnoDB;
```

接下来，使用 INSERT 命令向数据表插入数据。下列两个示例中，第一个 INSERT 语句将数据表的各列与值都罗列出来，而第二个 INSERT 语句中只给出了列值，并没有明确列出对应的数据列。使用第二种 INSERT 语句插入数据时，在 VALUES 之后一定要为数据表的所有数据列给出要存储的值。

```
mariadb [test]> INSERT INTO tab_test (fd1, fd2) VALUES (1, 'Matt');
Query OK, 1 row affected (0.02 sec)

mariadb [test]> INSERT INTO tab_test VALUES (2, 'Toto');
Query OK, 1 row affected (0.03 sec)
```

通常，使用第二种方式向数据表插入数据时，由于没有明确列出数据表的数据列，那么数据表结构发生变化后，执行时将极有可能引发错误。因此，开发应用程序时，建议各位采用第一种方式。

INSERT 语句经常使用 ON DUPLICATE KEY UPDATE 选项，启用该选项后，数据表中已经存在重复记录时，就会执行 UPDATE 操作，否则执行 INSERT 操作。

```
mariadb [test]> INSERT INTO tab_test(fd1, fd2)
                  VALUES (3, 'Lee') ON DUPLICATE KEY UPDATE fd2='Lee';
Query OK, 1 row affected (0.03 sec)

mariadb [test]> INSERT INTO tab_test(fd1, fd2)
                  VALUES (3, 'SeongUck') ON DUPLICATE KEY UPDATE fd2='SeongUck';
Query OK, 2 rows affected (0.02 sec)
```

执行第一条 INSERT 语句时，由于不存在与 fd1 列（主键）值重复的记录，所以 ON DUPLICATE KEY UPDATE 之后的部分会忽略，(3, 'Lee') 会保存到数据表。而执行第二条 INSERT 语句时，由于数据表中已经存在 fd1=3 的记录，所以 INSERT INTO … VALUES …部分忽略，将执行 UPDATE fd2='SeongUK'，将记录 (3, 'Lee') 更新为 (3, 'SeongUck')。

3.5.2 SELECT

使用 SELECT 语句可以检索数据表中存储的记录。下列示例使用 SELECT 语句显示存储在 tab_test 数据表中的所有记录。

```
mariadb [test]> SELECT * FROM tab_test;
+-----+----------+
| fd1 | fd2      |
+-----+----------+
|   1 | Matt     |
|   2 | Toto     |
|   3 | SeongUck |
+-----+----------+
3 rows in set (0.00 sec)
```

如果没有在 SELECT 语句中明确使用条件语句（WHERE），它会将数据表中的所有记录取出并显示。但数据表包含的记录很多时，使用它取出所有记录并返回给用户是不符合访问目的的。用户通常只对某些数据感兴趣，此时可以在 SELECT 语句中使用 WHERE 子句，只取出符合某些特定条件的记录。

```
mariadb [test]> SELECT * FROM tab_test WHERE fd1=1;
+-----+------+
| fd1 | fd2  |
+-----+------+
|   1 | Matt |
+-----+------+
1 row in set (0.00 sec)
```

上述两个示例中，SELECT 关键字之后全都使用了“*”，它表示数据表中的所有数据列。有时，因为数据表中的数据列太多，很难查找到要访问的数据，此时不要在 SELECT 之后使用“*”，只要将要访问的列名直接放到 SELECT 之后即可。

```
mariadb [test]> SELECT fd2 FROM tab_test;
+----------+
| fd2      |
+----------+
| Matt     |
| Toto     |
| SeongUck |
+----------+
3 rows in set (0.00 sec)
```

若使用 mysql 客户端连接数据库，查询时可以使用“\G”选项，按列输出数据表中的记录，如下所示。

```
mariadb [test]> SELECT * FROM tab_test\G
*************************** 1. row ***************************
fd1: 1
fd2: Matt
```

```
*************************** 2. row ***************************
fd1: 2
fd2: Toto
*************************** 3. row ***************************
fd1: 3
fd2: SeongUck
3 rows in set (0.00 sec)
```

请注意，在 SQL 语句中使用“\G”选项时，句末不要再使用分号（;），否则会引发错误。像这样，在 SQL 语句的末尾使用“\G”选项，mysql 客户端会按列输出获取的记录。

3.5.3 UPDATE

UPDATE 语句并非用于删除插入（INSERT）的记录，而是对记录的部分数据列进行修改。下面使用 UPDATE 语句对插入（INSERT）的记录进行修改。

```
mariadb [test]> UPDATE tab_test SET fd2='Brandon' WHERE fd1=1;
Query OK, 1 row affected (0.02 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mariadb [test]> SELECT * FROM tab_test WHERE fd1=1;
+------+---------+
| fd1  | fd2     |
+------+---------+
|    1 | Brandon |
+------+---------+
1 row in set (0.00 sec)
```

上述示例先从数据表中查找到 fd1 值为 1 的记录，然后将记录的 fd2 列值修改为 Brandon。使用 UPDATE 语句时，要特别注意 UPDATE 中不使用 WHERE 条件子句的情形。

```
mariadb [test]> UPDATE tab_test SET fd2='Brandon';
Query OK, 2 rows affected (0.02 sec)
Rows matched: 3  Changed: 2  Warnings: 0

mariadb [test]> SELECT * FROM tab_test;
+------+---------+
| fd1  | fd2     |
+------+---------+
|    1 | Brandon |
|    2 | Brandon |
|    3 | Brandon |
+------+---------+
3 rows in set (0.00 sec)
```

如示例所示，UPDATE 语句中不使用 WHERE 条件子句时，列值的修改将针对数据表中的所有记录。实际数据库环境中，若不小心对某个数据表进行了这种操作，造成的后果相当于丢失了数据表中的所有数据，请各位一定要注意。

3.5.4 REPLACE

前面讲过 INSERT INTO … ON DUPLICATE KEY UPDATE…语句，从表面上看，REPLACE 语句的处理过程与之类似。也就是说，指定的记录不存在时，执行 INSERT 操作，否则执行 UPDATE 操作。

```
mariadb [test]> REPLACE tab_test SET fd1=1, fd2='Matt';
Query OK, 2 rows affected (0.01 sec)

mariadb [test]> SELECT * FROM tab_test WHERE fd1=1;
+------+------+
| fd1  | fd2  |
+------+------+
|    1 | Matt |
+------+------+
1 row in set (0.00 sec)
```

由于 tab_test 数据表中存在 fd1 为 1 的记录，所以将进行 UPDATE 操作，仅将 fd2 的列值修改为“Matt”。但是，实际执行 REPLACE 语句时，若数据表中存在主键或唯一键重复的记录，会先删除记录，然后执行 INSERT 操作插入新记录。对用户而言，整个处理过程看上去与 UPDATE 非常相似，但实际上，REPLACE 语句执行前的记录与执行后的记录从内部来看是完全不同的。

数据表中存在重复记录时，执行 REPLACE 语句会先将记录删除，然后再次执行 INSERT 操作，所以与 INSERT INTO … ON DUPLICATE KEY UPDATE…语句相比，它造成的系统负荷更高，消耗的系统资源也更多。如果可能，请各位尽量使用 INSERT INTO … ON DUPLICATE KEY UPDATE…语句，而不要使用 REPLACE 语句。

3.5.5 DELETE

要删除数据表中已经存在的记录，使用 DELETE 语句即可。

```
mariadb [test]> DELETE FROM tab_test WHERE fd1=1;
Query OK, 1 row affected (0.02 sec)

mariadb [test]> SELECT * FROM tab_test;
+------+----------+
| fd1  | fd2      |
+------+----------+
```

```
|    2 | Brandon |
|    3 | Brandon |
+------+---------+
2 rows in set (0.00 sec)
```

与其他 SQL 语句类似，DELETE 语句中也可以使用 WHERE 子句，用以删除符合指定条件的记录。如果不使用 WHERE 条件子句，则表示 DELETE 操作针对数据表中的所有记录，即删除全部记录。

```
mariadb [test]> DELETE FROM tab_test;
Query OK, 2 rows affected (0.03 sec)

mariadb [test]> SELECT * FROM tab_test;
Empty set (0.00 sec)
```

MariaDB 或 MySQL 中，所有 SQL 语句默认以 AUTO-COMMIT（自动提交）模式执行。也就是说，SQL 语句成功执行后，内部会自动进行 COMMIT，修改的内容会永久保存。执行 DELETE 语句仅删除数据表的部分记录时，若没有使用 WHERE 条件子句，则将删除数据表中的所有记录，实际操作中会经常出现这种错误。当然，出现这种失误时，可以借助备份进行恢复。但如果没有事先备份或备份太旧，就会永远失去这些误删的数据。因此，使用 DELETE 或 UPDATE 语句时一定要注意，在实际的使用环境中，建议各位在关闭 AUTO-COMMIT 模式的状态下执行 DELETE 与 UPDATE 语句。这样，出现误删操作时还可能挽回，否则误删的数据就会永久丢失。

第4章

执行计划分析

4.1 概要 MariaDB MySQL

4.2 准备示例数据

4.3 执行计划分析 MariaDB MySQL

4.4 优化器提示 MariaDB MySQL

4.5 执行计划分析时的注意事项

无论做什么事，都需要事先制定计划。比如去国外旅行，首先要决定是经停还是直飞，然后选择从哪个机场出发，是仁川机场、金浦机场，还是金海机场？再决定乘坐何种交通工具从家前往机场。到达目的地后，还要选择各条旅游路线。即使去同一地，也可以选择不同的线路与方式。

MariaDB 服务器中，同一结果也可以使用不同方法获得，但使用这些方法所花费的代价是不同的，要从中选择一个最合适、花费最少的。旅行也一样，动身之前要先从网上或书上看一些相关信息，选择一种花费最少的方案，然后决定旅游线路。MariaDB 服务器中，为了实现以最优方式进行查询，要参考统计信息，这些统计信息记录着各数据表数据的分布情况。通过比较基本数据，得到最优的执行计划。包含 MariaDB 的大部分 DBMS 中，优化器负责该功能。

MariaDB 中，借助 EXPLAIN 命令可以查看查询执行计划，使用 EXPLAIN 命令会返回大量信息。本章将学习执行计划显示内容所代表的含义，详细介绍 MariaDB 服务器内部都做了哪些工作，最后再简单讨论执行计划的评价标准。

4.1 概要 MariaDB MySQL

无论哪种 DBMS，决定查询执行计划的优化器是最复杂的部分，而且由优化器创建的执行计划也是很难理解的。但是，只有理解了执行计划，才能找出执行计划不合理的部分，并通过最优方法确定一个更好的执行计划。分析执行计划之前，先学习几部分内容。

4.1.1 查询执行过程

MariaDB 服务器中，查询的执行过程大致分为如下 3 个阶段：

(1) 从用户接收 SQL 语句，切分语句并分析词法语法，最后生成 MariaDB 服务器可以理解的解析树；
(2) 确认 SQL 的解析信息，选择从哪个数据表读取，以及使用哪种索引读取数据表；
(3) 借助第二个阶段中确定的读表顺序或所选索引，从存储引擎获取数据。

第一个阶段通常称为 SQL 解析，由 MariaDB 服务器的 SQL 解析器负责处理。若 SQL 语句存在语法错误，则可在该阶段得到检测。经过 SQL 解析后，最终形成 SQL 解析树。MariaDB 服务器使用 SQL 解析树而非 SQL 语句本身以执行查询。

第二个阶段会依据第一个阶段形成的 SQL 解析树处理如下内容。

- 删除不必要的条件，将复合运算简单化。
- 若存在多表连接，则确定读表顺序。
- 根据用于各数据表的条件与索引统计信息，确定要使用的索引。
- 将获取的记录放入临时表，确定是否需要再次加工。

当然，此外还要进行许多其他处理，但以上这些处理是最具代表性的。第二个阶段会进行优化处理并确定最终执行计划，这是由 MariaDB 服务器的优化器负责的。第二个阶段完成后，

会得到查询执行计划。

第三个阶段将依据执行计划向存储引擎请求读取记录，MariaDB 引擎会对从存储引擎获得的记录进行连接、排序等处理。

其中，第一个阶段与第二个阶段几乎都是由 MariaDB 引擎处理的，第三个阶段由 MariaDB 引擎与存储引擎同时参与处理。图 4-1 显示了 SQL 解析器与优化器在 MariaDB 整个体系结构中的位置。

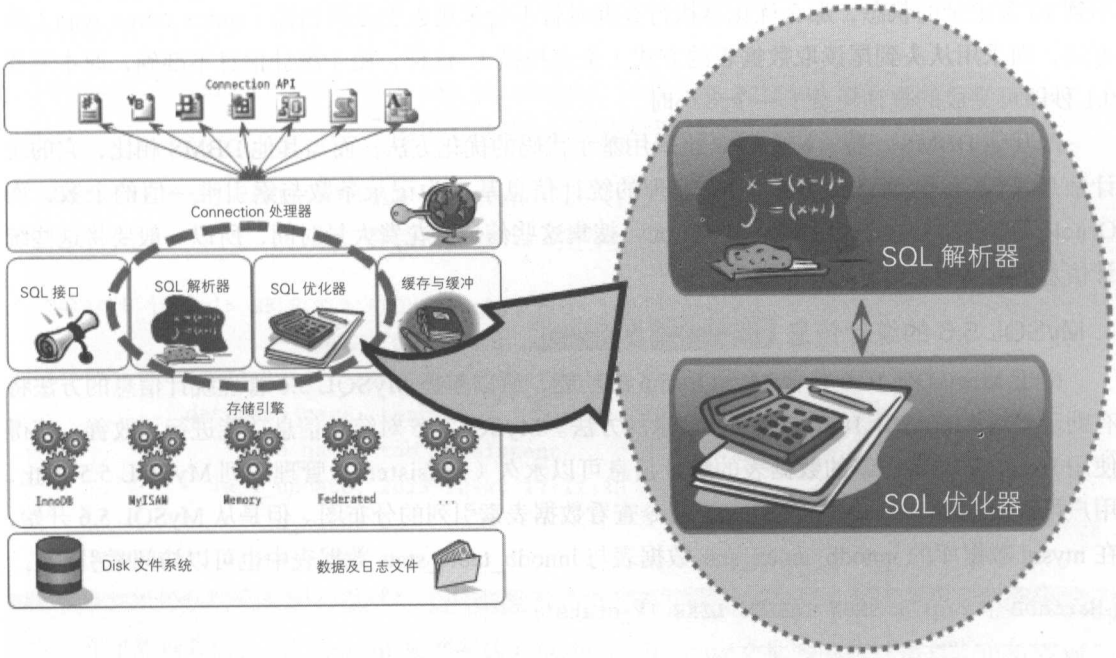


图 4-1 查询解析与优化器

4.1.2 优化器种类

从作用而言，优化器相当于数据库服务器的大脑，它大致分为两大类：一类是现在大部分 DBMS 所用的“基于代价的优化器”（Cost-Based Optimizer，CBO），另一类是 Oracle 以前大量使用的“基于规则的优化器”（Rule-Based Optimizer，RBO）。

(1) “基于规则的优化”默认不考虑目标数据表的记录数、选择度等，而根据优化器中内置的若干规则制定相应的执行计划。该方式中，制定执行计划时不会调查统计信息（数据表的记录数或列值分布图），所以对于相同查询，其执行方法几乎总是相同的。但是，大部分 DBMS 很早以前就开始不支持基于规则的优化了，而且一直没有更新。

(2) 基于代价的优化”处理查询时会创建多种可用方法，然后根据各种处理的消耗信息与目标数据表的统计信息，计算各执行计划代价，最后从中选择代价最小的执行计划作为最终执行计划。

基于规则的优化方法中，几乎没有各数据表或索引的统计信息，相对较慢的 CPU 计算会导

致代价计算过程负荷较重，现在几乎不再使用。目前，大部分 RDBMS 几乎都采用基于代价的优化器，MariaDB 是如此。

4.1.3 统计信息

基于代价的优化中，统计信息是最重要的依据。若统计信息不准确，执行查询就可能走向完全错误的方向。比如一个拥有 1 亿条记录的数据表，若其统计信息未及时更新，仍然保持着不到 10 条记录的状态，那么优化器执行查询时就不会采用索引范围扫描（index range scan）的方法，而采用从头到尾读取数据表的方式（全表扫描）。这样，由于统计信息不准确，原本需要 0.1 秒即可完成的查询耗费了一个多小时。

跟其他 DBMS 一样，MariaDB 也采用基于代码的优化方法；而与其他 DBMS 相比，它的统计信息又没有那么多样。MariaDB 管理的统计信息基本为记录条数与索引唯一值的个数，而 Oracle 等 DBMS 中的统计信息相当庞大，搜集这些信息要花费大量时间，所以一般要将这些统计信息单独备份。

1. MySQL 5.6 的统计信息 MariaDB MySQL

由于 MariaDB 10.0 包含 MySQL 5.6 的功能，所以掌握 MySQL 5.6 管理统计信息的方法将有助于理解 MariaDB 10.0 管理统计信息的方法。MySQL 5.6 对统计信息功能进行了改善，使得使用 InnoDB 存储引擎的数据表的统计信息可以永久（Persistent）管理。到 MySQL 5.5 为止，用户可以使用 SHOW INDEX STATS 命令查看数据表索引列的分布图。但是从 MySQL 5.6 开始，在 mysql 数据库的 innodb_index_stas 数据表与 innodb_table_stats 数据表中也可以访问索引。

```
MariaDB [mysql]> SHOW TABLES LIKE '%_stats';
+---------------------------+
| Tables_in_mysql (%_stats) |
+---------------------------+
| innodb_index_stats        |
| innodb_table_stats        |
+---------------------------+
```

在 MySQL 5.6 中创建数据表时，可以设置 STATS_PERSISTENT 选项，根据其设置值确定是否以数据表为单位永久保存统计信息。

```
mysql> CREATE TABLE tab_test (fd1 INT, fd2 VARCHAR(20), PRIMARY KEY(fd1))
                    ENIGNE=InnoDB STATS_PERSISTENT={ DEFAULT | 0 | 1 }
```

STATS_PERSISTENT=0

采用 MySQL 5.5 以前的方式管理数据表的统计信息，不保存于 mysql 数据库的 innodb_index_stas 与 innodb_table_stats 数据表。

STATS_PERSISTENT=1

将数据表的统计信息保存到 mysql 数据库的 innodb_index_stas 与 innodb_table_stats 数据表。

STATS_PERSISTENT=DEFAULT

其效果与创建数据表时不设置 STATS_PERSISTENT 选项相同，是否永久管理数据表的统计信息由 innodb_stats_presistent 系统变量设置值决定。

innodb_stats_presistent 系统变量的默认值为 ON（1），创建数据表时，若不使用 STATS_PERSISTENT 选项，则使用永久统计信息，并将统计信息保存到 innodb_index_stas 与 innodb_table_stats 数据表。下列示例创建 tab_persistent 与 tab_transient 两个数据表时，使用了 STATS_PERSISTENT 选项，并且分别设置为 1 与 0。

```
MariaDB [test]> CREATE TABLE tab_persistent (fd1 INT PRIMARY KEY, fd2 INT)
                        ENGINE=InnoDB STATS_PERSISTENT=1;

MariaDB [test]> CREATE TABLE tab_transient (fd1 INT PRIMARY KEY, fd2 INT)
                        ENGINE=InnoDB STATS_PERSISTENT=0;

MariaDB [test]> SELECT * FROM mysql.innodb_table_stats
                      WHERE table_name IN ('tab_persistent', 'tab_transient')\G
*************************** 1. row ***************************
           database_name: test
              table_name: tab_persistent
             last_update: 2013-12-28 17:11:30
                  n_rows: 0
    clustered_index_size: 1
sum_of_other_index_sizes: 0
```

创建数据表后，访问 mysql 数据库的 innodb_table_stats 数据表的统计信息。可以看到，只能访问 tab_persistent 数据表（采用 STATS_PERSISTENT=1 选项创建）的统计信息。当然，也可以使用 ALTER TABLE 命令将数据表的统计信息更改为永久状态（STATS_PERSISTENT=1）或短期状态（STATS_PERSISTENT=0）。

到 MySQL 5.5 或 MariaDB 5.5 为止，每当遇到下列情形，就会重新收集数据表的统计信息。

- 新打开数据表时
- 大量修改数据表的记录时（在数据表的全部记录中执行 1/16 的 UPDATE、INSERT、DELETE 时）
- 执行 ANALYZE TABLE 命令时
- 执行 SHOW TABLE STATUS 或 SHOW INDEX FROM 命令时
- 激活 InnoDB 监视器时
- innodb_stats_on_metadata 系统设置为 ON 并执行 SHOW TABLE STATUS 命令时
- 其他

若数据表的统计信息经常变化，原本一直使用索引范围扫描处理应用程序查询的 MariaDB 服务器某一天会突然开始采用全表扫描方式处理查询。但是，引入永久统计信息后，可以有效防止意外修改统计信息。另外，将 innodb_stats_auto_recalc 系统变量设置为 OFF 也可以防止自

动收集统计信息。innodb_stats_auto_recalc 系统变量的默认值为 ON，若想使用永久统计信息，需要将其设置为 OFF。此外，创建数据表时可以使用 STATS_AUTO_RECALC 选项，以表为单位进行调整，决定是否自动收集统计信息。

STATS_AUTO_RECALC=1

以 MySQL 5.5 之前的方式自动收集数据表的统计信息。

STATS_AUTO_RECALC=0

只在执行 ANALYZE TABLE 命令时收集数据表统计信息

STATS_AUTO_RECALC=DEFAULT

效果与创建数据表时不使用 STATS_AUTO_RECALC 选项一样。是否收集数据表的统计信息由 innodb_stats_auto_recalc 系统变量的设置值决定。

MySQL 5.5 提供了 innodb_stats_sample_pages 系统变量，收集数据表的统计信息时，它决定对几个 InnoDB 数据表块进行采样。但从 MySQL 5.6 开始不再支持（Deprecated）该变量，取而代之的是 innodb_stats_transient_sample_pages 与 innodb_stats_persistent_sample_pages 两个系统变量。

innodb_stats_transient_sample_pages

该系统变量的默认值为 8，即收集统计信息时默认采取 8 个页块的样本数据进行分析，并将分析结果用作统计信息。

innodb_stats_persistent_sample_pages

默认值为 20，执行 ANALYZE TABLE 命令时将随意采取 20 个页面进行分析，并将分析结果保存到永久统计信息数据表以供使用。

若想使用永久统计信息，可以在 MariaDB 服务器维护或用量不多的时间段收集，这样得到的统计信息会更准确。当然，要收集更准确的统计信息一定会耗费大量时间，但由于统计信息的准确性决定着查询性能，所以投入这些时间是值得的。若想进一步提高统计信息收集的准确性，只要将 innodb_stats_persistent_sample_pages 变量值设置得大一些即可。但也不要设置得太高，否则会大大增加收集统计信息的时间。

2. MariaDB 10.0 的统计信息 MariaDB

MariaDB 5.5 版本之前，所有数据表的统计信息都由相关数据表的存储引擎进行管理。但依据不同存储引擎、以不同形态管理数据表的统计信息，可能出现如下问题。

- 各存储引擎提供的统计信息内容相当不足。
- 向 MySQL（MariaDB）存储引擎接口传递统计信息时有诸多限制（不允许收集非索引列的统计信息）。
- 不易控制统计信息，统计信息不永久，短期（Transient）管理。

由于存在这些问题，MySQL 5.5 或 5.6 不允许对非索引的数据列收集统计信息。为了解决这些问题，MariaDB 10.0 提供了综合统计信息管理功能，使用时不受存储引擎限制。从 MariaDB 10.0 开始，不仅可以管理非索引列的统计信息，而且这些综合管理的统计信息也可以

永久使用。因此，MariaDB 也可以像其他 RDBMS 一样，将具有较高准确度的统计信息单独备份，或者将其再次恢复使用。统计信息不准确时，管理员可以直接修改，这样 MariaDB 优化器就可以用更准确的信息制定更好的执行计划。

MariaDB 的综合统计信息由 table_stats、column_stats、index_stats 数据表进行管理，这些数据表位于 MariaDB 服务器的默认数据库 mysql，并且都使用 MyISAM 存储引擎。

```
MariaDB> SHOW TABLES LIKE '%_stats';
+---------------------------+
| Tables_in_mysql (%_stats) |
+---------------------------+
| column_stats              |
| index_stats               |
| table_stats               |
+---------------------------+
3 rows in set (0.00 sec)
```

综合统计信息表的各数据列存储着如下信息。

table_stats

数据列	含义
db_name	目标数据表所属的数据库名
table_name	目标数据表名
cardinality	数据表的记录数

column_stats

数据列	含义
db_name	目标数据表所属的数据库名
table_name	目标数据表名
column_name	目标数据列名
min_value	相应列的最小值（整数类型也以字符串格式保存）
max_value	相应列的最大值（整数类型也以字符串格式保存）
nulls_ratio	NULL 值的比率（0：无 NULL；0.5：拥有 NULL 值的记录 50%；1：所有记录为 NULL）
avg_length	列值的平均字节数
avg_frequency	拥有重复值的平均记录数（1：无重复值）

index_stats

数据列	含义
db_name	目标数据表所属的数据库名
table_name	目标数据表名
index_name	目标索引名
prefix_arity	索引键区号 INDEX(fd1, fd2) 时，若 prefix_arity=1，则对 fd1 列统计；若 prefix_arity=2，则对 (fd1, fd2) 统计
avg_frequency	拥有重复值平均记录数（1：无重复值）

前面已经讲解了 MySQL 5.6 中的两种统计信息管理方法（永久、短期），与此不同的是，MariaDB 10.0 引入了与存储引擎无关的综合型统计信息，并提供了三种管理方法。事实上，MariaDB 10.0 的综合统计信息与 MySQL 5.6 的永久统计信息是同时使用的，执行查询时，系统变量 use_stat_tables 设置值决定使用哪种统计信息，其取值为 never、complementary、preferably 三者之一，默认值为 never。

use_stat_tables='never'

采用 MySQL 5.6 的统计信息管理方式，统计信息不会收集到 mysql 数据库的 table_stats、column_stats、index_stats 数据表。MariaDB 10.0 也将 use_stat_tables 设置为 never，直接使用 MySQL 5.6 的永久统计信息。

use_stat_tables='complementary'

将系统变量 use_stat_tables 设置为 complementary 后，优先使用各存储引擎提供的统计信息。但存储引擎未提供信息或提供的信息不足时，就会使用 MariaDB 的综合统计信息。若数据表使用的是 MySQL 5.6 的 InnoDB 存储引擎，则有关于数据表全部记录数与索引列基数（Cardinality）信息，但非索引列的统计信息不会得到管理。此外，若使用 MyISAM 或 MEMORY 存储引擎，则不会有任何统计信息。为了克服这一弱点，可以使用 complementary 模式，该模式将使用 MariaDB 的综合统计信息。

use_stat_tables='preferably'

将系统变量 use_stat_tables 设置为 preferably 后，MariaDB 引擎会优先使用 MariaDB 的综合统计信息，然后再使用各存储引擎管理的统计信息。若 MariaDB 综合统计信息不存在，则使用存储引擎提供的统计信息制定查询执行计划。

将系统变量 innodb_stats_auto_recalc 设置为 ON 之后，数据表结构发生变化或有大量数据变化时，MariaDB 10.0 的 InnoDB 存储引擎会自动收集统计信息。但此时收集的统计信息与 MariaDB 的综合统计信息无关，仅用于更新 InnoDB 存储引擎的统计信息。请看下列示例。

```
MariaDB [test]> SHOW GLOBAL VARIABLES 'innodb_stats_auto_recalc';
+--------------------------+-------+
| Variable_name            | Value |
+--------------------------+-------+
| innodb_stats_auto_recalc | ON    |
+--------------------------+-------+

MariaDB [test]> INSERT INTO tab_recalc SELECT * FROM employees;
Query OK, 300024 rows affected (4.35 sec)
Records: 300024  Duplicates: 0  Warnings: 0

MariaDB [test]> SELECT * FROM mysql.table_stats WHERE table_name='tab_recalc';
Empty set (0.00 sec)

MariaDB [test]> SELECT * FROM mysql.innodb_table_stats WHERE table_name='tab_recalc';
```

```
*************************** 1. row ***************************
           database_name: test
              table_name: tab_recalc
             last_update: 2013-12-28 19:13:29
                 n_rows: 299645
    clustered_index_size: 929
sum_of_other_index_sizes: 0
```

上述示例创建了名为 tab_recalc 的空表，然后从 employees 数据表检出（SELECT）三十多万条记录插入（INSERT）其中。执行查询后，查看 MariaDB 的综合统计表（mysql.table_stats）与 InnoDB 的永久统计表（mysql.innodb_table_stats），可以发现 MariaDB 的综合统计表不存在任何信息，而 InnoDB 的统计表中存在相应统计信息。若将系统变量 innodb_stats_auto_recalc 设置为 OFF，则 InnoDB 的统计表不会保存任何值。

由上可知，MariaDB 10.0 的综合统计信息不会自动收集。若想收集，则要执行 ANALYZE TABLE 命令。

```
MariaDB [test]> ANALYZE TABLE tab_recalc;
+-----------------+---------+----------+----------+
| Table           | Op      | Msg_type | Msg_text |
+-----------------+---------+----------+----------+
| test.tab_recalc | analyze | status   | OK       |
+-----------------+---------+----------+----------+

MariaDB [test]> SELECT * FROM mysql.table_stats WHERE table_name='tab_recalc';
+---------+------------+-------------+
| db_name | table_name | cardinality |
+---------+------------+-------------+
| test    | tab_recalc |      300024 |
+---------+------------+-------------+

MariaDB [test]> SELECT * FROM mysql.innodb_table_stats WHERE table_name='tab_recalc'\G
*************************** 1. row ***************************
           database_name: test
              table_name: tab_recalc
             last_update: 2013-12-28 19:23:33
                 n_rows: 299246
    clustered_index_size: 929
sum_of_other_index_sizes: 0
```

上述示例执行 ANALYZE TABLE 命令后，查看 MariaDB 的综合统计表，可以看到其变化情况。tab_recalc 数据表的基数（Cardinality）保存为 300 024，其实这是个非常准确的数据。由于 MariaDB 10.0 的 ANALYZE TABLE 命令执行时会进行全表扫描与索引全扫描，所以最后收

集到的数据是相当准确的。

注意 令人遗憾的是，MariaDB 10.0.8 中，只要执行 ANALYZE TABLE 命令就必须进行全表扫描或索引全扫描。因此，MariaDB 的综合统计信息收集起来会相当消耗资源，使用时一定要小心。

实际应用环境中，使用 MariaDB 时通常会架设主 / 从服务器。主 MariaDB 服务器主要用于向外提供服务，很难收集统计信息时，可以通过从服务器收集统计信息，然后复制给主 MariaDB 服务器。这也是一种比较好的方法。

系统变量 use_stat_tables 设置为 never 的状态下，在 MariaDB 10.0 中执行 ANALYZE TABLE 命令后，会一直收集存储引擎级别的统计信息，但不会收集 MariaDB 的综合统计信息。而将 use_stat_tables 设置为 complementary 或 preferably 时，执行 ANALYZE TABLE 命令后，会同时收集存储引擎级别的统计信息和 MariaDB 的综合统计信息。

此外，MariaDB 10.0 对 ANALYZE TABLE 命令的语法进行了一些扩充，以更好地适应综合统计信息的收集工作。

```
## 对 tbl 数据表收集统计信息，且仅针对 col1、col2 数据列以及 idx1、idx2
ANALYZE TABLE tbl PERSISTENT FOR COLUMNS (col1,col2) INDEXES (idx1,idx2);

## 对 tbl 数据表收集统计信息，且仅针对 col1、col2 数据列
ANALYZE TABLE tbl PERSISTENT FOR COLUMNS (col1,col2) INDEXES ( );

## 对 tbl 数据表收集统计信息，且仅针对 idx1、idx2
ANALYZE TABLE tbl PERSISTENT FOR COLUMNS ( ) INDEXES (idx1,idx2);

## 仅收集 tbl 数据表的统计信息
ANALYZE TABLE tbl PERSISTENT FOR COLUMNS ( ) INDEXES ( );

## 对数据表、所有列、所有索引收集索引统计信息
## 效果与 ANALYZE TABLE tbl 命令相同
ANALYZE TABLE tbl PERSISTENT FOR ALL;
```

4.1.4 直方图统计信息 MariaDB

直方图也是一种统计信息，用于分析列值的分布情况。MySQL 5.6 并不支持直方图，只有 MariaDB 10.0 支持。不使用直方图的情况下，数据库系统会依据一个数据列中唯一值的个数推测数据的大致分布情况，以此计算执行计划的代价。但这也只针对创建了索引的数据列，因为只有这种数据列的唯一值才会得到管理，而对于未建索引的数据列则很难做出准确预测。

1. 直方图

MySQL 5.5 或 MariaDB 5.5 中，预测执行计划是相当不准确的。为了解决这一问题，需要分析实际数据页。简单查询语句的执行计划中，可以将记录数（执行计划 rows 数据列的输出值）相当准确地预测出来。但在复杂的查询语句中，预测的记录数误差往往很大，常常导致选

择了错误的执行计划。

MariaDB 10.0 中，由索引形成的数据列以及未建索引的数据列都可以保存直方图信息。数据表所有列的最小值与最大值，以及拥有 NULL 值的记录比率、列值分布等都由直方图进行收集，然后在 mysql 数据库的 column_stats 数据表中管理。

```
MariaDB [employees]> DESC mysql.column_stats;
+--------------+---------------------------------+------+-----+---------+-------+
| Field        | Type                            | Null | Key | Default | Extra |
+--------------+---------------------------------+------+-----+---------+-------+
| db_name      | varchar(64)                     | NO   | PRI | NULL    |       |
| table_name   | varchar(64)                     | NO   | PRI | NULL    |       |
| column_name  | varchar(64)                     | NO   | PRI | NULL    |       |
| min_value    | varchar(64)                     | YES  |     | NULL    |       |
| max_value    | varchar(64)                     | YES  |     | NULL    |       |
| nulls_ratio  | decimal(12,4)                   | YES  |     | NULL    |       |
| avg_length   | decimal(12,4)                   | YES  |     | NULL    |       |
| avg_frequency| decimal(12,4)                   | YES  |     | NULL    |       |
| hist_size    | tinyint(3) unsigned             | YES  |     | NULL    |       |
| hist_type    | enum('SINGLE_PREC_HB','DOUBLE_PREC_HB') | YES  |     | NULL    |       |
| histogram    | varbinary(255)                  | YES  |     | NULL    |       |
+--------------+---------------------------------+------+-----+---------+-------+
```

MariaDB 10.0 在管理直方图时采用“高度均衡直方图”（Height-Balanced Histogram）算法，比其他商用 RDBMS 略有落后。

“高度均衡直方图”算法先对所有列值进行排序，再按相同的记录个数将其划为几个组，然后取各组的最后一个值（排序后的最大值）保存到直方图。

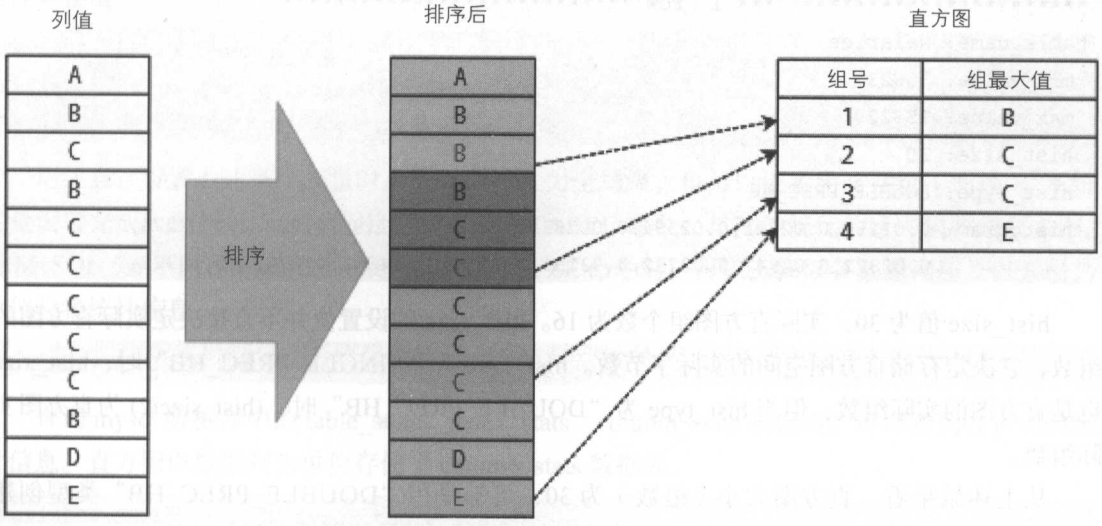


图 4-2 MariaDB 中使用的直方图算法

图 4-2 描述了 MariaDB 中使用的“高度均衡直方图”创建过程。创建“高度均衡直方图”时，先对数据表中的所有列值进行排序，然后按一定个数进行分组，再将组号与各组最大值保存到直方图。图 4-2 中，每 3 个列值被放入一组，总共形成 4 个分组。MariaDB 中，分组个数由系统变量 histogram_size 确定。当然，分组个数越多，直方图就越准确，但这需要更多的存储空间与分析时间。从图中可以看到，最后创建出的直方图中，组 2 与组 3 拥有相同的值“C”，由此可以获知某个特定列值分布有多广。

MariaDB 会为这样创建出的直方图值按不同组别各分配一个字节，然后存储到 VARBINARY(255) 类型的 histogram 数据列中。其实，真正存储的值并不是组的最大值，而是先计算“组最大值 /（列最大值 – 列最小值）* α”，然后将结果值依次存入 histogram 数据列的各字节。计算公式中，“α”将“组最大值 /（列最大值 – 列最小值）”的结果值（带有小数点）转换为整数，以便存储到 histogram 的 VARBINARY 数据类型。此时，“α”的值越大，直方图的准确度就越高，这也意味着需要更多存储空间。MariaDB 中有名为 histogram_type 的系统变量，其值为“SINGLE_PREC_HB”或“DOUBLE_PREC_HB”。取为“SINGLE_PREC_HB”时，“α”值为 28；取为“DOUBLE_PREC_HB”时，“α”值为 216。优化器使用直方图时，会逆向使用上述计算公式求解。

分析 mysql 数据库中的 column_stats 数据表后，可以直接查看 MariaDB 计算并存储的直方图。由于 histogram 数据列为 VARBINARY 数据类型，所以很难直接进行查看。为了解决这一问题，MariaDB 提供了 DECODE_HISTOGRAM 函数。

```
MariaDB> SELECT table_name, min_value, max_value,
                DECODE_HISTOGRAM(histogram, hist_type) as histogram
         FROM mysql.column_stats
         WHERE table_name='salaries' AND column_name='salary'\G
*************************** 1. row ***************************
table_name: salaries
 min_value: 38623
 max_value: 158220
 hist_size: 30
 hist_type: DOUBLE_PREC_HB
 histogram: 0.02759,0.02521,0.02394,0.02263,0.02202,0.02196,0.02222,0.02274,
            0.02377,0.02547,0.02792,0.03194,0.03830,0.04883,0.07225,0.54322
```

hist_size 值为 30，实际直方图组个数为 16。hist_size 的设置值并不直接决定实际直方图的组数，它决定存储直方图空间的实际字节数。hist_type 为“SINGLE_PREC_HB”时，hist_size 也是直方图的实际组数；但当 hist_type 为“DOUBLE_PREC_HB”时，(hist_size/2) 为直方图实际组数。

从上述结果看，直方图大小（组数）为 30，可知是用“DOUBLE_PREC_HB”类型创建的。如下查询 mysql.table_stats 数据表，可以看到 employees 数据表的全部记录数。

```
MariaDB> SELECT * FROM mysql.table_stats WHERE table_name='employees';
+-----------+------------+-------------+
| db_name   | table_name | cardinality |
+-----------+------------+-------------+
| employees | salaries   |     2844047 |
+-----------+------------+-------------+
```

也就是说，MariaDB 服务器创建直方图时，每个分组包含“2844047/16=177752.93”条记录。接下来，借助从 mysql.column_stats 数据表查询得到的 histogram 值的比率，查看直方图灵活应用的准确程度。

```
MariaDB [employees]> SELECT COUNT(*) FROM salaries WHERE salary<=38623+(158220-38623)*0.02759;
+----------+
| COUNT(*) |
+----------+
|   177733 |
+----------+
```

执行查询后，得到的结果为 177733。可以看到，它与初次创建直方图时各组中的记录数 177752.93 非常近似。

2. 在 MariaDB 中使用直方图

使用 ANALYZE TABLE 命令可以同时创建其他统计信息和直方图。是否收集直方图由系统变量 histogram_size 的取值确定。histogram_size 系统变量的默认值为 0，表示不使用直方图。若想使用直方图，首先要将 histogram_size 系统变量设置为大于 0 的值。此外，还要根据需要将系统变量 histogram_type 的值设置为 DOUBLE_PREC_HB 或 SINGLE_PREC_HB，以确定直方图的准确程度。

```
MariaDB> SET histogram_size=20;
MariaDB> SET use_stat_tables='preferably';
MariaDB> SET histogram_type='double_prec_hb';
```

请注意：设置上述系统变量时，既可以设置为全局级，也可以设置为会话级（连接）。系统变量设置完成后，如下所示，使用 ANALYZE TABLE 命令，与其他统计信息一起收集直方图。与 MySQL 5.6 不同，MariaDB 10.0 执行 ANALYZE 命令时，可以选择以各数据列还是以索引为单位收集统计信息。

```
MariaDB> ANALYZE TABLE salaries;
```

查询 mysql 数据库中的 table_stats、index_stats、column_stats 数据表，可以看到收集到的统计信息。直方图以数据列为单位存储于 column_stats 数据表。

```
MariaDB> SELECT * FROM mysql.table_stats;
```

```
MariaDB> SELECT * FROM mysql.index_stats;
MariaDB> SELECT * FROM mysql.column_stats;
```

优化器使用收集的直方图时，可以在下面这些不同级别层次上混合使用 MariaDB 引擎级别与各存储引擎级别的统计信息。optimizer_use_condition_selectivity 系统变量的默认值为 1。

optimizer_use_condition_selectivity=1

使用 MariaDB 5.5 中的选择度（Selectivity）预测方式。

optimizer_use_condition_selectivity=2

仅针对创建了索引的数据列的条件判断选择度。

optimizer_use_condition_selectivity=3

对所有数据列的条件判断选择度（不使用直方图）。

optimizer_use_condition_selectivity=4

对所有数据列的条件判断选择度（使用直方图）。

optimizer_use_condition_selectivity=5

对所有数据列的条件使用直方图判断选择度，对不能进行索引范围扫描的数据列采用记录采样以判断选择度。

提示 MariaDB 服务器中，优化器制定查询执行计划时，任意访问实际数据表的几条记录即可预测选择度。比如，类似 first_name LIKE'%sla' 的条件，不仅不能使用索引，而且连该条件选择度的好坏也无法判断。但针对 first_name LIKE'%sla' 这样的条件，MariaDB 会任意访问数据表的几条记录，以对该条件的选择度进行判断。此时，MariaDB 优化器会实际读取几条记录以预测选择度，取样条数由 optimizer_selectivity_sampling_limit 系统变量决定。只有 optimizer_use_condition_selectivity 系统变量值为 5 时，才会使用 optimizer_selectivity _sampling_limit 系统变量。

利用收集到的直方图信息，对未建索引的数据列也可以大致预测其记录条数。

```
MariaDB> SET use_stat_tables='preferably'
MariaDB> SET optimizer_use_condition_selectivity=4;
MariaDB> EXPLAIN EXTENDED SELECT * FROM salaries WHERE to_date<='1989-03-01';
```

上述 salaries 数据表的执行计划如下所示，需要注意 rows 与 filter 两个域，分析该两个域值的方法如下：

(1) InnoDB 存储引擎读取并返回全部 2844047 条记录；

(2) MariaDB 引擎从上一步读取的记录中选择符合 WHERE 条件（to_date<='1989-03-01'）的记录并返回给用户。

最终要返回给用户的记录数为 111486，即总记录条数 2844047 的 3.92%[①]。

① 关于理解 EXPLAIN EXTENDED 执行计划中 filtered 列值的方法，后面的执行计划部分会进行详细讲解。

Id	select_type	table	type	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	salaries	ALL	NULL	NULL	NULL	2844047	3.92	Using where

下面执行查询，看看实际返回的记录数究竟是多少。

```
MariaDB> SELECT COUNT(*) FROM salaries WHERE to_date<='1989-03-01';
+----------+
| count(*) |
+----------+
|   125536 |
+----------+
```

如上所示，查询得到的实际记录数为 125536 条，与查询执行计划中计算的记录数 111486 几乎一样。若要得到上述示例中的查询执行计划，需要先将 histogram_size 设置为 100、histogram_type 设置为 DOUBLE_PREC_HB，然后再执行 ANALYZE TABLE salaries 命令。

4.1.5 连接优化器选项

MariaDB 中，有两个优化器选项用于优化连接查询的执行计划。事实上，这些选项并非只存在于 MariaDB，它们也同样存在于 MySQL 5.0 之后的版本。虽然它们很重要，但可能很多人并不懂，下面一起进行讲解。

MariaDB 与 MySQL 分别对连接优化功能做了很多改进，但随着数据表数的增多，查找最优执行计划变得相当困难。一个查询中连接的表数增多，将导致制定执行计划花费更多时间。若数据表数超过了某个限制，制定执行计划所需的时间就会大大增加，有时需要几个小时甚至几天。为什么会出现这种现象？如何避免这种现象？下面为大家进行分析。

MariaDB 提供了两种制定最优连接执行计划的算法：

(1) Exhaustive 检索

(2) Heuristic 检索（Greedy 检索）

两个算法名称都含有“检索”（Search）字样，我认为可能是它们都需要从 *N* 个连接对象数据表中查找最优连接顺序组合的原因。Exhaustive 检索是 MySQL 5.0 及其以前版本使用的连接优化方法，它会针对 FROM 子句指定的所有数据表的组合计算执行计划的代价，然后从中选择最优的一个。使用该方法时，若数据表有 10 个，那么可能的连接组合为 20!(Factorial, 3628800)。因此，数据表超过 10 个时，使用 Exhaustive 检索方法制定执行计划将花费几分钟。并且，即使数据表只增加一个，即从 10 个变为 11 个，也多需要 11 倍的时间。

Heuristic 检索又称 Greedy 检索，它是从 MySQL 5.0 开始引入的连接优化方法，旨在解决使用 Exhaustive 检索方法的耗时问题。Greedy 检索采用比 Exhaustive 检索更复杂的方式以确定最优连接顺序。

(1) 全部 *N* 个数据表中，通过 optimizer_search_depth 系统变量指定个数的数据表创建可能的连接组合。

(2) 在步骤 1 中创建的连接组合中，选择一个代价最小的执行计划。

(3) 将步骤 2 中选定的执行计划的第一个数据表选作“部分执行计划”的第一个数据表。

(4) 全部 *N*-1 个数据表中（不含步骤 3 中选择的数据表），通过 optimizer_search_depth 系统变量指定个数的数据表创建可能的连接组合。

(5) 将步骤 4 中创建的连接组合逐个代入步骤 3 中创建的“部分执行计划”，计算执行代价。

(6) 从步骤 5 中选择一个代价最小的执行计划，从中选择第二个数据表作为步骤 3 中创建的“部分执行计划”的第二个数据表。

(7) 重复步骤 4~6，直到无剩余数据表，并将数据表的连接顺序记录到“部分执行计划”。

(8) 最后，“部分执行计划”被确定为数据表的连接顺序。

使用 Greedy 检索方法可以根据 optimizer_search_depth 系统变量的值大大减少连接优化花费的代价。optimizer_search_depth 系统变量的默认值为 64。Greedy 检索中，对于外部连接这种已经有确定连接顺序的内容也进行优化。64 是个不小的数值，即使数据表只有 20 个，制定执行计划也要耗费相当长的时间。

MariaDB 与 MySQL 提供了两个用于连接优化的系统变量，分别为 optimizer_prune_level、optimizer_search_depth。

optimizer_prune_level

该系统变量用于指定要使用哪种连接优化搜索算法。若 optimizer_prune_level 设置为 OFF，MariaDB 将采用 Exhaustive 检索算法进行连接优化；若其值设置为 ON，则使用 Greedy 检索算法。optimizer_prune_level 的默认值为 ON，如果 MariaDB 中有很多数据表要频繁执行连接查询，建议不要修改 optimizer_prune_level 值。

optimizer_search_depth

optimizer_search_depth 系统变量仅用于 Greedy 搜索。根据其设置值不同，制定连接查询执行计划的速度也有快有慢。当然，若查询中不怎么需要表连接，或者查询中需要连接的表不多，那么其值对制定查询执行计划的性能影响不大。[①]optimizer_search_depth 的默认值为 64，其实这是个很大的值。其值越小，MariaDB 所选执行计划的优化度越低。

那么，表连接很多时，制定查询执行计划会有多慢？调整 optimizer_prune_level 与 optimizer_search_depth 后，又会变得有多快？下面通过示例进行分析。

首先创建 30 个数据表，表名从 tab01 到 tab30，它们都具有相似的表结构，即拥有两个数据列、一个主键、一个辅助索引。然后分别向各数据表插入 2000 条记录。为了让表结构与索引结构更多样化，创建某些表时，要用 INT、BIGINT 数据类型生成 fd1、fd2 数据列；而对于有些表，则故意删除其辅助索引。并且使用从 MariaDB 10.0 开始可用的 Sequence 存储引擎，向各表插入 2000 条记录。

```
CREATE TABLE tab01 (
  fd1 char(20) NOT NULL,
```

① 若连接的表为 3~5 个，在 Exhaustive 检索中也只要分析检查 120 多个执行计划，所以处理速度还是很快的。

```
  fd2 char(20) DEFAULT NULL,
  PRIMARY KEY (fd1),
  KEY ix_fd2 (fd2)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

INSERT INTO tab01 SELECT seq, seq FROM seq_1_to_2000;
```

像这样，将用于测试的数据表准备好后，接下来要借助查询查看执行计划。由于查询未经优化，所以建议各位不要实际使用该查询语句查询数据。虽然查询本身未被优化，但是通过查看查询执行计划观察 MariaDB 制定执行计划所需的时间已经足够了，而不必关心实际的查询性能。

```
MariaDB> SET SESSION optimizer_prune_level = 1;
MariaDB> SET SESSION optimizer_search_depth= { 1 | 5 | 10 | 15 | 20 | 25 |
 30 | 35 | 40 | 62 };
MariaDB> EXPLAIN
    SELECT *
    FROM
        tab01, tab02, tab03, tab04, tab05,
        tab06, tab07, tab08, tab09, tab10,
        tab11, tab12, tab13, tab14, tab15,
        tab16, tab17, tab18, tab19, tab20,
        tab21, tab22, tab23, tab24, tab25,
        tab26, tab27, tab28, tab29, tab30
    WHERE
        tab01.fd1=tab02.fd1
        AND tab02.fd1=tab03.fd2
        AND tab03.fd1=tab04.fd2
        AND tab04.fd2=tab05.fd1
        AND tab05.fd2=tab06.fd1
        AND tab06.fd2=tab07.fd2
        AND tab07.fd1=tab08.fd1
        AND tab08.fd2=tab09.fd1
        AND tab09.fd1=tab10.fd2
        AND tab10.fd1=tab11.fd2
        AND tab11.fd2=tab12.fd1
        AND tab12.fd2=tab13.fd2
        AND tab13.fd1=tab14.fd1
        AND tab14.fd2=tab15.fd1
        AND tab15.fd1=tab16.fd2
        AND tab16.fd1=tab17.fd1
        AND tab17.fd2=tab18.fd2
        AND tab18.fd1=tab19.fd1
        AND tab19.fd2=tab20.fd2
```

```
        AND tab20.fd1=tab21.fd1
        AND tab21.fd2=tab22.fd2
        AND tab22.fd2=tab23.fd1
        AND tab23.fd1=tab24.fd2
        AND tab24.fd2=tab25.fd2
        AND tab25.fd1=tab26.fd2
        AND tab26.fd1=tab27.fd2
        AND tab27.fd2=tab28.fd1
        AND tab28.fd2=tab29.fd1
        AND tab29.fd2=tab30.fd2;
```

图 4-3 描述了为连接 30 个数据表的查询语句制定执行计划所耗费的时间，单位为毫秒，其中横轴为 optimizer_search_depth 的取值，取值范围为 1~62，每次取值增 5。

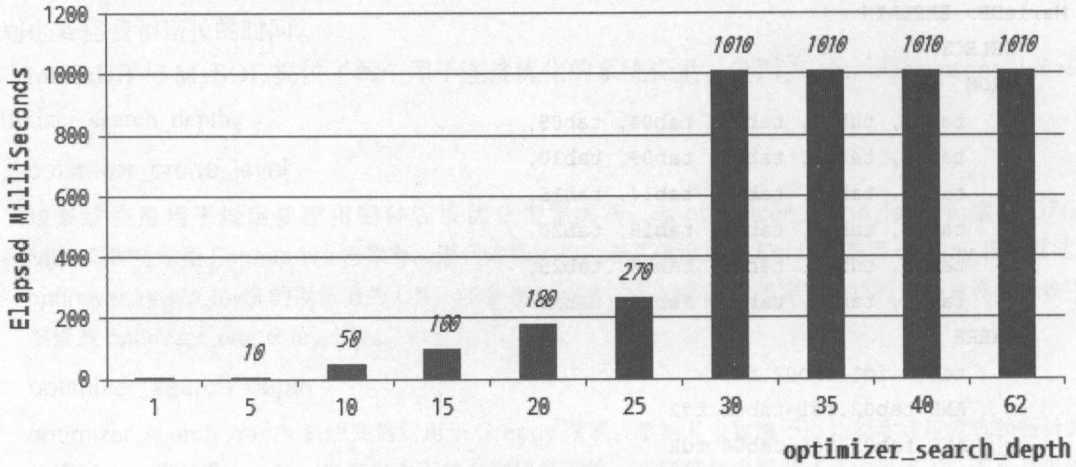


图 4-3 optimizer_search_depth 与制定执行计划的时间

我的测试中，optimizer_search_depth 为 1 时，需要 0.4 秒左右，该值可能有较大误差，出于相对比较的需要，图中故意略去。从图中可以看到，随着 optimizer_search_depth 取值的增大，制定查询执行计划的所需时间逐渐增加。此外，由于查询中需要连接 30 个数据表，所以 optimizer_search_depth 值达到 30 以上时，它对制作执行计划的性能影响就不明显了。

提示 各位若想在自己的系统中查看制定查询计划处理得有多快，可以使用 MariaDB 的 CPU Profiling 功能。

```
MariaDB> SET PROFILING=1;
MariaDB> SELECT * FROM tab_test WHERE fd=1;

MariaDB> SHOW PROFILES;
+----------+------------+----------------------------------+
| Query_ID | Duration   | Query                            |
```

```
+----------+------------+-----------------------------------+
|        1 | 0.05178391 | SELECT * FROM tab_test WHERE fd=1 |
+----------+------------+-----------------------------------+

MariaDB> SHOW PROFILE CPU FOR QUERY 1;
+----------------------+----------+----------+------------+
| Status               | Duration | CPU_user | CPU_system |
+----------------------+----------+----------+------------+
| starting             | 0.000092 | 0.000000 |   0.000000 |
| checking permissions | 0.000011 | 0.000000 |   0.000000 |
| Opening tables       | 0.009897 | 0.000000 |   0.000000 |
| System lock          | 0.000012 | 0.000000 |   0.000000 |
| Table lock           | 0.000015 | 0.000000 |   0.000000 |
| Opening tables       | 0.017150 | 0.000000 |   0.000000 |
| init                 | 0.000048 | 0.000000 |   0.000000 |
| optimizing           | 0.000027 | 0.000000 |   0.000000 |
| statistics           | 0.024340 | 0.015600 |   0.000000 |
| preparing            | 0.000046 | 0.000000 |   0.000000 |
| executing            | 0.000022 | 0.000000 |   0.000000 |
| query end            | 0.000012 | 0.000000 |   0.000000 |
| closing tables       | 0.000012 | 0.000000 |   0.000000 |
| freeing items        | 0.000011 | 0.000000 |   0.000000 |
| updating status      | 0.000069 | 0.000000 |   0.000000 |
| cleaning up          | 0.000020 | 0.000000 |   0.000000 |
+----------------------+----------+----------+------------+
```

执行 SHOW PROFILE CPU FOR QUERY 1 命令所返回的结果中，显示在 statistics 项之后的时间即为制定查询执行计划所耗费的时间。上述示例中，该时间为 0.024 秒。MariaDB 中，数据表的统计信息也存储于数据表。初次执行查询时，要读取统计信息表，所以性能会略慢一些。对相同 SQL 语句执行多次并观察 Profile 信息，即可大致把握其性能。

此外，在 Linux 系统中使用 SHOW PROFILE CPU 命令会引起 rusage() 系统调用，rusage() 系统调用以进程而非线程为单位进行。因此，SHOW PROFILE CPU 命令的结果可能会有同一进程其他线程的干涉，请不要忘记这一点。

假设在实际服务环境中使用上述查询，只制定查询执行计划就大约要耗费 1 秒钟。如果获得实际查询结果的过程中要从磁盘或内存获取数据，再加上运算耗费的时间，查询的执行性能就会大大降低。此外还需要注意，MariaDB 与 MySQL 中几乎不会重复使用查询执行计划[①]，这样每次执行查询时都要重新为查询制定执行计划。如果每次都要花 1 秒钟，那么频繁执行这样

① 仅在使用预定义语句（Prepare statement）情形下，会话级别的执行计划才会得到重复使用。除此之外，MariaDB 与 MySQL 中几乎不会共享执行计划。请注意，在预定义语句中重复使用会话级别的执行计划时，若连接和要执行的查询有很多，则要占用的内存使用量就会大幅增加。

的查询时，就会给系统带来相当大的负担。

图 4-3 的测试将 optimizer_prune_level 设置为 1，即使用 Greedy 检索算法。此时只需调整 optimizer_search_depth 的取值，即可观察其取值对制定执行计划所需时间的影响。若一个查询需要连接 30 个数据表，将 optimizer_prune_level 设置为 0，使用 Exhaustive 检索算法会怎样呢？由于要连接 30 个数据表，所以要做比较判断的的连接计划总数为 30!(Factorial)。30! 有多大，各位可以用计算器算一下。若将 optimizer_prune_level 设置为 0，使用 Exhaustive 检索算法，那么只要数据表超过 10 个，就不可能查看查询的执行计划了。

网上有文章称，若一个查询中要连接的数据表少于 4~5 个，则可以使用 Exhaustive 检索算法；若超过 4~5 个，则建议使用 Greedy 检索算法。但每次执行查询时，系统设置（optimizer_search_depth 与 optimizer_prune_level 既是全局级变量也是会话级变量，可以只针对某个特定连接进行设置）都会发生变化，查询的执行就变得难以控制。查询执行过程中若不会发生什么特殊问题，建议将 optimizer_prune_level 设置为 1，即采用 Greedy 检索算法（MariaDB 与 MySQL 服务器的默认设置）。

若要连接的数据表很多而导致制定查询执行计划很耗时，可以考虑将 optimizer_search_depth 降低为 10，再应用到 MariaDB 服务器。若想对查询进行调优，则要先确定最优执行计划，再使用类似 STRAIGHT_JOIN 的优化器提示，将连接顺序强制应用至优化器。

4.2 准备示例数据

详细查看执行计划前，首先要准备用于执行查询的示例数据。若各位所用的数据与我测试用的不同，则很可能得到与本书示例不同的结果。即便是相同数据，若数据表的统计信息不同，得到的查询执行计划也会有所区别。

4.2.1 加载示例数据

本书介绍的查询语句或执行计划绝大部分基于 employees 数据库，各位可以访问下列网站进行下载。

https://launchpad.net/test-db/

但从上述网站下载的测试数据库与我想要的还有些不同，为了让它满足查询测试的要求，需要对其进行修改或添加数据。因此，我将上述网站中的 employees 数据库的数据文件和单独编写的 SQL 文件上传到我的个人博客。若各位确实需要示例数据文件的原始文件，可以直接从上述网站下载。

将数据文件加载到 MariaDB 服务器之前，需要大家先设置自己的 MariaDB 服务器，使之与我所用的测试环境一致。若在不同环境中测试本书示例，则我所说的执行计划或优化内容可能不会表现出来。在 MariaDB 服务器的设置文件（my.cnf 或 my.ini）中，对影响执行计划的设置

进行如下修改。

```
## Linux 系列操作系统中（不含 Windows）
## 忽略 Oracle InnoDB，用 Percona XtraDB 存储引擎
ignore_builtin_innodb = ON
plugin-load = ha_xtradb.so

innodb_stats_auto_recalc=0
use_stat_tables='preferably'
innodb_stats_transient_sample_pages=8
innodb_stats_persistent_sample_pages=20
innodb_stats_persistent=1
innodb_stats_on_metadata=0
innodb_page_size=16K
innodb_file_per_table=1
innodb_file_format=Antelope
innodb_buffer_pool_size=64MB
```

如上所示，设置文件修改完后，重启 MariaDB 服务器。

下面访问我的博客（http://cafe.naver.com/realmysql），从“资料室”下载示例数据库文件。正常完成下载后，按如下步骤将数据加载到 MariaDB 数据库。

- 下载示例数据文件并解压缩；
- 在 Windows 或 Linux 命令窗口（Linux Shell）中转到解压缩目录；
- 在解压缩目录下启动 mysql 客户端；
- 执行如下命令并加载示例数据；

```
MariaDB>SOURCE employees.sql
```

- 执行下列命令，删除不必要的索引及新数据表与索引。

```
MariaDB>SOURCE employees_schema_modification.sql
```

上述第 4 个步骤中，使用 Launchpad 网站（https://launchpad.net/test-db/）提供的示例数据，创建并加载 employees 数据库。第 5 个步骤中，从 Launchpad 网站提供的数据中删除不必要的外键，并创建索引与数据表。

到步骤 5 为止，所需要的示例数据准备完成。启动 MariaDB 客户端，登录相应数据库，使用以下命令列出数据库中的所有数据表。

```
MariaDB> USE employees;

MariaDB> SHOW TABLES;
+----------------------+
| Tables_in_employees |
+----------------------+
```

```
| departments          |
| dept_emp             |
| dept_manager         |
| employee_name        |
| employees            |
| salaries             |
| tb_dual              |
| titles               |
+----------------------+
```

4.2.2 收集统计信息

将数据加载到MariaDB的过程中，不会自动应用示例数据库数据表的统计信息。从MariaDB 10.0开始，统计信息的内容变化很大，不会像MariaDB 5.x一样将统计信息自动更改为与用户意图无关的内容。

运行MariaDB客户端程序，执行如下ANALYZE命令，收集统计信息。

```
MariaDB> USE employees;

MariaDB> ANALYZE TABLE departments  ;
MariaDB> ANALYZE TABLE dept_emp     ;
MariaDB> ANALYZE TABLE dept_manager ;
MariaDB> ANALYZE TABLE employee_name;
MariaDB> ANALYZE TABLE employees    ;
MariaDB> ANALYZE TABLE salaries     ;
MariaDB> ANALYZE TABLE tb_dual      ;
MariaDB> ANALYZE TABLE titles       ;
```

若存储统计信息的数据表尚未准备好或误删，则执行ANALYZE命令收集统计信息的过程中会显示如下错误信息。

```
MariaDB [employees]> analyze table employees;
+--------------------+---------+----------+----------------------------------------+
| Table              | Op      | Msg_type | Msg_text                               |
+--------------------+---------+----------+----------------------------------------+
| employees.employees | analyze | Error    | Table'mysql.table_stats'doesn't exist  |
| employees.employees | analyze | status   | OK                                     |
+--------------------+---------+----------+----------------------------------------+
```

退出MariaDB客户端，执行mysql_upgrade命令，重新创建存储统计信息的数据表。即使数据库已经更新过，也可以使用“--force”选项强制进行更新，生成基本数据表。

```
shell> mysql_upgrade --force -h127.0.0.1 -P3306 uroot -p
```

注意 MariaDB 与 MySQL 中，优化各种类型的查询时，设置 optimizer_switch 系统变量可以控制要使用或不使用哪些选项。根据 optimizer_switch 系统变量的设置，查询执行计划也会变得非常不同。首先阅读本书中有关执行计划的内容，对执行计划略微熟悉后，再学习优化开关及其引起的不同执行计划的相关内容。

本书所用的 optimizer_switch 选项设置如下所示，这些选项的设置值在 MariaDB 10.0.6 中是默认设置值。由于这些选项的设置会导致执行计划不同，各位若想得到与本书示例相同的执行计划，请先检查各自的 optimizer_switch 设置是否与下列设置一致。[①]

```
index_merge=on
index_merge_union=on
index_merge_sort_union=on
index_merge_intersection=on
index_merge_sort_intersection=off
engine_condition_pushdown=off
index_condition_pushdown=on
derived_merge=on
derived_with_keys=on
firstmatch=on
loosescan=on
materialization=on
in_to_exists=on
semijoin=on
partial_match_rowid_merge=on
partial_match_table_scan=on
subquery_cache=on
mrr=off
mrr_cost_based=off
mrr_sort_keys=off
outer_join_with_cache=on
semijoin_with_cache=on
join_cache_incremental=on
join_cache_hashed=on
join_cache_bka=on
optimize_join_buffer_size=off
table_elimination=on
extended_keys=off
exists_to_in=off
```

如果对逐个查看 optimizer_switch 选项感到厌烦，可以使用如下命令将 optimizer_switch 的所有项设置为默认值。但使用下列命令修改后，重启 MariaDB 服务器时，这些设置将会消失。运行本书示例时，要将显式修改 optimizer_switch 的设置完全从 MariaDB 服务器的设置文件（my.cnf 或 my.ini）中删除。

① 若示例中需要修改 optimizer_switch 设置，将会单独给出修改 optimizer_switch 选项的命令。

```
MariaDB> SET GLOBAL optimizer_switch=DEFAULT;
MariaDB> SET SESSION optimizer_switch=DEFAULT;
```

与其他系统变量不同，用户可以逐个修改 optimizer_switch 选项。虽然多个设置值全部存储在同一 optimizer_switch 系统变量，但用户仍然能够修改其中某一个选项值，而不影响其他选项的设置值。

```
MariaDB> SHOW GLOBAL VARIABLES LIKE 'optimizer_switch'\G
*************************** 1. row ***************************
Variable_name: optimizer_switch
        Value: index_merge=off,index_merge_union=on,index_merge_sort_union=on,...

MariaDB> SET GLOBAL optimizer_switch='index_merge=on';

MariaDB> SHOW GLOBAL VARIABLES LIKE 'optimizer_switch'\G
*************************** 1. row ***************************
Variable_name: optimizer_switch
        Value: index_merge=on,index_merge_union=on,index_merge_sort_union=on,...
```

4.3 执行计划分析 MariaDB MySQL

MariaDB 中，可以使用 EXPLAIN 命令查看查询执行计划。使用 EXPLAIN 命令时，若不附加任何选项，则只显示默认查询执行计划。若使用 EXPLAIN EXTENDED 或 EXPLAIN PARTITIONS 命令，则能查看更详细的执行计划。使用附加选项时，在默认执行计划的基础上还会再附加 1 个信息。下面先使用不带选项的 EXPLAIN 命令得到执行计划，再对其进行详细分析，以正确理解默认执行计划，然后再使用带 PARTITIONS 或 EXTENDED 选项的命令查看执行计划。

如下所示，使用 EXPLAIN 命令时，只要将要查看的 SELECT 查询语句置于 EXPLAIN 关键字之后即可。执行 EXPLAIN 命令后，返回一个包含执行计划相关信息的表格。执行计划中存在类似 possible_keys 这类内容较长的项目，这些项目几乎没什么用。本书只讲解与执行计划密切相关的项目，略去其他无关紧要的项目，且执行计划中值为 NULL 的部分全部用空白表示。

```
EXPLAIN
SELECT e.emp_no, e.first_name, s.from_date, s.salary
FROM employees e, salaries s
WHERE e.emp_no=s.emp_no
LIMIT 10;
```

执行 EXPLAIN 计划后，根据查询语句的特征，以表格形式显示多于 1 行的结果。表中行数（记录）与查询语句中用到的数据表（也包含用子查询创建的临时数据表）数一致。执行顺序自上而下依次显示（若有 UNION 或相关子查询，可能不会依次显示）。输出的执行计划中，越显示在上方的结果（id 列值越小）越为查询的外部（Outer）或先访问的数据表，越显示在下

方的结果（id 列值越大）越为查询的内部（Inner）或后访问的数据表。但与查询语句直接比较时，从执行计划的顶部开始与数据表进行匹配，这样会比较容易理解。

id	select_type	table	type	Key	key_len	ref	rows	Extra
1	SIMPLE	e	index	ix_firstname	44		300584	Using index
1	SIMPLE	s	ref	PRIMARY	4	employees.e.emp_no	4	

与其他 DBMS 不同，为了根据需要形成执行计划，MariaDB 有时会直接执行部分查询。因此，查询本身相当复杂时，访问执行计划可能会变得很慢。此外，也无法查看 UPDATE、INSERT、DELETE 语句的执行计划。若想查看 UPDATE、INSERT、DELETE 语句的执行计划，可以先将其转换为带 WHERE 条件的 SELECT 语句，进行初步掌握。

那么，执行计划中，各数据列的含义是什么？哪些值会显示到各数据列呢？

4.3.1 id 列

一个 SELECT 语句可以包含多个子查询语句（SUB SELECT）。请看下列查询语句。

```
SELECT ...
FROM (SELECT ... FROM tb_test1) tb1,
  tb_test2 tb2
WHERE tb1.id=tb2.id;
```

上述查询语句包含子查询语句，可以将它们单独分离出来，如下所示。像这样，下面将以 SELECT 关键字为单位进行的区分表述为“单位（SELECT）查询”。

```
SELECT ... FROM tb_test1;
SELECT ... FROM tb1, tb_test2 tb2 WHERE tb1.id=tb2.id;
```

执行计划中，最左侧的 id 列用于识别每一个单位 SELECT 查询。就本示例的查询而言，执行计划中最少含 2 个 id 值。

若一个 SELECT 语句中要连接多个数据表，执行计划记录会根据连接的表数进行显示，并赋予相同的 id。下面示例中只有一个 SELECT 语句，但需要连接多个表，此时 id 值不会增加，拥有相同 id。

```
EXPLAIN
SELECT e.emp_no, e.first_name, s.from_date, s.salary
FROM employees e, salaries s
WHERE e.emp_no=s.emp_no
LIMIT 10;
```

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	s	index	ix_salary	4	NULL	2844047	Using index
1	SIMPLE	e	eq_ref	PRIMARY	4	employees.s.emp_no	1	

相反，下列查询执行计划中，整个查询语句由3个“单位SELECT查询”组成，所以执行计划的各记录拥有不同id值。

```
EXPLAIN
SELECT
( (SELECT COUNT(*) FROM employees) + (SELECT COUNT(*) FROM departments) ) AS total_count;
```

id	select_type	table	type	key	key_len	ref	rows	Extra
1	PRIMARY	NULL	NULL	NULL	NULL	NULL	NULL	No tables used
3	SUBQUERY	departments	index	PRIMARY	12	NULL	9	Using index
2	SUBQUERY	employees	index	ix_hiredate	3	NULL	300024	Using index

4.3.2 select_type列

select_type数据列指明各“单位SELECT查询”的查询类型，select_type数据列的列值如下所示。

1. SIMPLE

进行不需要UNION操作或不含子查询的简单SELECT查询时，相应查询语句的select_type即为SIMPLE（查询中包含连接的情形也一样）。无论查询语句多么复杂，执行计划中select_type为SIMPLE的单位查询一定只有1个。最外侧SELECT查询的select_type通常为SIMPLE。

2. PRIMARY

一个需要UNION操作或含子查询的SELECT查询执行计划中，位于最外层的单位查询的select_type即为PRIMARY。与SIMPLE一样，select_type为PRIMARY的单位SELECT查询也只存在1个，位于查询最外侧的SELECT单位查询的select_type即为PRIMARY。

3. UNION

由UNION操作联合而成的单位SELECT查询中，除第一个外，第二个以后的所有单位SELECT查询的select_type都为UNION。UNION的第一个单位SELECT的select_type不是UNION，而是DERIVED。它是一个临时表，用于存储联合（UNION）后的查询结果。

```
EXPLAIN
SELECT * FROM (
  (SELECT emp_no FROM employees e1 LIMIT 10)
  UNION ALL
  (SELECT emp_no FROM employees e2 LIMIT 10)
  UNION ALL
  (SELECT emp_no FROM employees e3 LIMIT 10)
) tb;
```

以上查询的执行计划如下所示。3 个进行联合（UNION）的单位 SELECT 查询中，只有第一个（e1 数据表）不是 UNION，其余两个的 select_type 均为 UNION。UNION 的第一个查询设置为代表整个 UNION 结果的 select_type 类型。此处要将 3 个子查询的结果用 UNION ALL 进行联合，并创建临时表进行使用，所以 UNION ALL 的第一个查询的 select_type 设置为 DERIVED。

id	select_type	table	type	key	key_len	ref	rows	Extra
1	PRIMARY	<derived2>	ALL	NULL	NULL	NULL	30	
2	DERIVED	e1	index	ix_hiredate	3	NULL	300024	Using index
3	UNION	e2	index	ix_hiredate	3	NULL	300024	Using index
4	UNION	e3	index	ix_hiredate	3	NULL	300024	Using index
NULL	UNION RESULT	<union2,3,4>	ALL	NULL	NULL	NULL	NULL	

4. DEPENDENT UNION

与 UNION select_type 一样，DEPENDENT UNION 出现在 UNION 或 UNION ALL 形成的集合查询中。此处的 DEPENDENT 表示 UNION 或 UNION ALL 联合而成的单位查询受外部影响。下列示例查询中，两个 SELECT 查询用 UNION 联合起来，所以 UNION 出现在 select_type 中。从 IN 所包含的子查询中可以看到，两个查询通过 UNION 连接在一起。MariaDB 中，不会在默认优化器模式下先处理 IN(subquery) 查询内部的子查询，而是先读取外部的 employees 数据表，再执行子查询。employees 数据表的列值会对子查询产生影响。像这样，引用外部值处理内部查询时，DEPENDENT 关键字就会出现在 select_type 中。

```
EXPLAIN
SELECT *
FROM employees e1
WHERE e1.emp_no IN (
  SELECT e2.emp_no FROM employees e2 WHERE e2.first_name='Matt'
  UNION
  SELECT e3.emp_no FROM employees e3 WHERE e3.last_name='Matt'
);
```

最后，内部“e2.emp_no=e1.emp_no”与“e3.emp_no=e1.emp_no”条件会自动添加到 UNION 所用的 SELECT 查询的 WHERE 条件，然后再执行。由于外部定义的 employees 数据表的 emp_no 数据列要在子查询中使用，所以 DEPENDENT UNION 关键字出现在 select_type 中。

id	select_type	table	type	key	key_len	ref	rows	Extra
1	PRIMARY	e1	ALL	NULL	NULL	NULL	300024	Using where
2	DEPENDENT SUBQUERY	e2	eq_ref	PRIMARY	4	func	1	Using where
3	DEPENDENT UNION	e3	eq_ref	PRIMARY	4	func	1	Using where
NULL	UNION RESULT	<union2,3>	ALL	NULL	NULL	NULL	NULL	

注意 一个单位 SELECT 查询中还包含其他单位 SELECT 时，我们将内部包含的 SELECT 称为子查询。一个查询中含有子查询时，子查询通常会比外部查询先执行，这种处理方式的速度一般会非常快。但对于 select_type 包含 DEPENDENT 关键字的子查询，由于它要依赖外部查询，所以绝对不会比外部查询先执行。因此，select_type 包含 DEPENDENT 关键字的子查询执行往往会比较低效。

5. UNION RESULT

UNION RESULT 为包含 UNION 结果的数据表。MariaDB 中，UNION ALL 或 UNION（DISTINCT）查询会将所有 UNION 结果创建为临时表。执行计划中，该临时表所在行的 select_type 为 UNION RESULT。由于 UNION RESULT 在实际查询中不是单位查询，所以没有单独的 id 值。

```
EXPLAIN
SELECT emp_no FROM salaries WHERE salary>100000
UNION ALL
SELECT emp_no FROM dept_emp WHERE from_date>'2001-01-01';
```

id	select_type	Table	Type	key	key_len	ref	rows	Extra
1	PRIMARY	salaries	range	ix_salary	4	NULL	193362	Using where; Using index
2	UNION	dept_emp	index	ix_fromdate	3	NULL	1	Using where; Using index
NULL	UNION RESULT	<union1,2>	ALL	NULL	NULL	NULL	NULL	

上述执行计划的最后一行（UNION RESULT）的 table 列显示为“<union1, 2>”，表示它是 id 分别为 1 和 2 的单位查询的查询结果的联合，如图 4-4 所示。

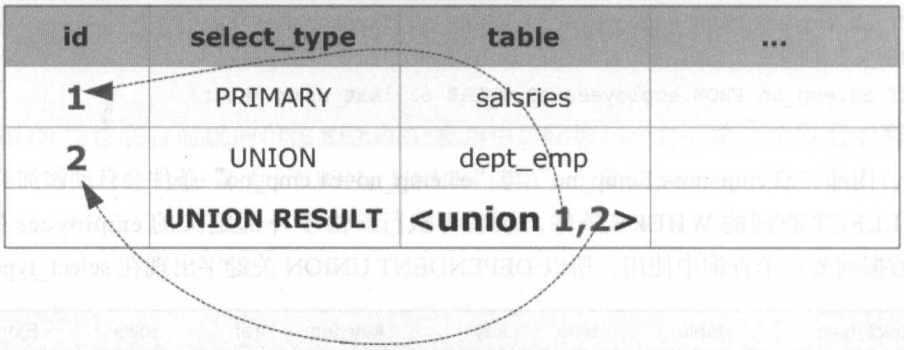


图 4-4 UNION RESULT 的 <union N, M> 含义

6. SUBQUERY

通常的子查询含义较为丰富，而此处的 SUBQUERY 仅指除 FROM 子句以外使用的子查询。

```
EXPLAIN
```

```
SELECT
  e.first_name,
  (SELECT COUNT(*) FROM dept_emp de, dept_manager dm WHERE dm.dept_no=de.dept_no)
AS cnt FROM employees e
WHERE e.emp_no=10001;
```

id	select_type	table	type	key	key_len	ref	rows	Extra
1	PRIMARY	e	const	PRIMARY	4	const	1	
2	SUBQUERY	de	index	PRIMARY	16	NULL	1	Using index
2	SUBQUERY	dm	ref	PRIMARY	12	employees.de.dept_no	1	Using index

MariaDB 的执行计划中，FROM 子句使用的子查询的 select_type 标为 DERIVED。除此之外，其他位置使用的子查询全部标注为 SUBQUERY。请注意，本书或 MariaDB 手册中所说的“派生表”与 DERIVED 具有相同含义。

提示 所用位置不同，子查询拥有不同称谓。

嵌套查询（Nested Query）

用于 SELECT 列的子查询称为嵌套查询。

子查询（Sub Query）

一般将用于 WHERE 子句的查询称为子查询。

派生表（Derived）

MariaDB 将 FROM 子句中的子查询称为派生表，在 RDBMS 中也称为内联视图（Inline View）或嵌套 SELECT。

此外，也可以根据子查询返回值的特点进行区分。

标量子查询（Scalar SubQuery）

只返回单个值（单行单例）的子查询。

单行子查询（Row Sub Query）

与列数无关，只返回单行的子查询。

7. DEPENDENT SUBQUERY

若子查询要使用外部 SELECT 查询中定义的列，则称此子查询为 DEPENDENT SUBQUERY。请看下列示例查询。

```
EXPLAIN
SELECT e.first_name,
  (SELECT COUNT(*)
   FROM dept_emp de, dept_manager dm
   WHERE dm.dept_no=de.dept_no AND de.emp_no=e.emp_no) AS cnt
FROM employees e
WHERE e.first_name='Matt';
```

如示例所示，由于内部子查询的结果要依赖外部 SELECT 查询的列，所以使用 DEPENDENT 关键字。与 DEPENDENT UNION 类似，DEPENDENT SUBQUERY 也先执行外部查询，再执行内部查询（子查询），所以它往往比一般子查询（不带 DEPENDENT 关键字）的处理速度要慢。

id	select_type	table	type	key	key_len	ref	rows	Extra
1	PRIMARY	e	ALL	NULL	NULL	NULL	300024	Using where
2	DEPENDENT SUBQUERY	de	ref	ix_empno_fromdate	4	employees.e.emp_no	1	Using index
2	DEPENDENT SUBQUERY	dm	ref	PRIMARY	12	employees.de.dept_no	1	Using index

8. DERIVED

MySQL 5.5 或 MariaDB 5.2 中，为 FROM 子句中的子查询创建执行计划时，所建计划的 select_type 总是 DERIVED。但从 MariaDB 5.3 与 MySQL 5.6 开始，根据优化器选项（optimizer_switch），可能会将 FROM 子句中的子查询与外部查询综合起来进行优化。DERIVED 意味着要为单位 SELECT 查询的执行结果在内存或磁盘上创建临时数据表。select_type 为 DERIVED 时创建的临时数据表也称为派生表。MySQL 5.5 与 MariaDB 5.2 中，派生表无任何索引，与其他表连接时通常会产生性能问题。但从 MariaDB 5.3 开始，可以根据优化器选项、查询的特征等向创建的临时表添加索引。

```
EXPLAIN
SELECT *
FROM
  (SELECT de.emp_no FROM dept_emp de GROUP BY de.emp_no) tb,
  employees e
WHERE e.emp_no=tb.emp_no;
```

执行上述查询时会删除 FROM 语句的子查询，然后用连接进行处理。其实，其他 DBMS 也提供了这种用于重新生成查询的优化功能。但从下列执行计划可以看出，MariaDB 会将 FROM 语句中的子查询创建为临时表进行处理。

id	select_type	table	type	key	key_len	ref	rows	Extra
1	PRIMARY	<derived2>	ALL	NULL	NULL	NULL	2	
1	PRIMARY	e	eq_ref	PRIMARY	4	tb.emp_no	1	
2	DERIVED	de	index	ix_empno_fromdate	7	NULL	1	Using index

从 MySQL 5.0 开始，针对连接查询的优化技术已经非常成熟。如果各位还在使用无法对派生表进行有效优化的 MySQL 或 MariaDB 版本，若可能，建议用连接处理 DERIVED 形态的执

行计划。从 MariaDB 10.0 开始，FROM 中子查询的优化有了很大改善。若有可能，尽量将不必要的子查询用连接重组后再处理。但像以前一样，优化器可以处理的操作很有限，所以编写优化的查询是非常重要的。

注意 为查询调优而查看执行计划时，首先要看 select_type 的列值是否为 DERIVED。有时会因为别无他法而必须使用子查询，但只要能使用连接进行处理，我强烈建议各位使用连接而非子查询。实际开发查询时，一般采用逐步增加功能的方式。这种开发过程中编写的大部分查询采用子查询方式，而非连接方式。当然，采用这种方式会提高开发效率，但查询性能较差。如果编写查询时采用子查询，则会比较方便。各位可以使用子查询，但最后一定要记得将子查询转换为连接。这样养成习惯后，编写查询时就不会再用子查询，而会直接使用连接编写复杂查询了。

9. UNCACHEABLE SUBQUERY

即使一个查询语句中只有一个子查询，这也并不意味着子查询实际只执行一次。但执行相同条件的子查询时，前一次子查询的结果会存储到内部缓存空间，后面相同条件的子查询就不再执行，而直接使用前一次查询的结果。此处所说的子查询缓存功能与查询缓存或派生表完全无关，请注意区分。下面简单比较 SUBQUERY 与 DEPENDENT SUBQUERY 使用缓存的方法。

- SUBQUERY 不受外层查询影响，只在开始执行一次，并将结果缓存起来，每次需要时使用缓存中的结果。
- DEPENDENT SUBQUERY 以所依赖的外部查询的列值为单位进行缓存并使用。

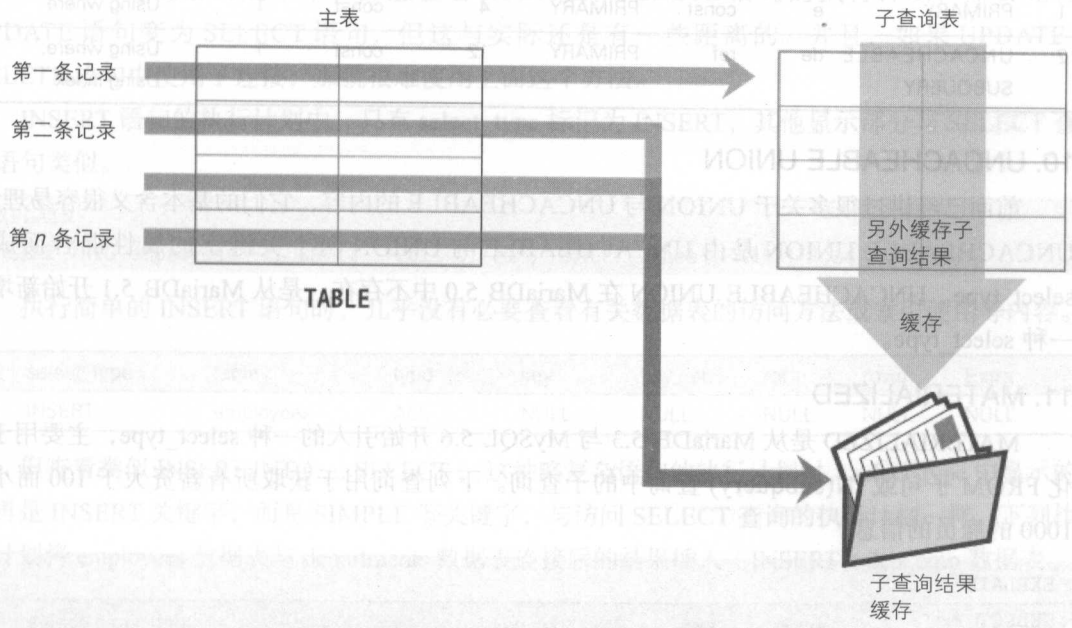


图 4-5 SUBQUERY 与 DEPENDENT SUBQUERY 的结果缓存

图 4-5 描述了 select_type 为 SUBQUERY 时的缓存使用方法。从图中可知，缓存仅在开始时

创建了1次。但处理DEPENDENT SUBQUERY时，子查询的结果并非仅缓存1次，而以外部查询的值为单位创建缓存（即依次反复执行上图过程）。

select_type为SUBQUERY时，“UNCACHEABLE SUBQUERY”是否可用缓存会造成结果差异。有时可能会因为子查询中有如下一些情形而使缓存本身无法使用，此时，select_type标记为UNCACHEABLE SUBQUERY。下面是几种无法使用缓存的情形。

- 子查询含有用户变量时
- 子查询含有NOT-DETERMINISTIC属性的存储例程时
- 子查询使用每次调用有不同结果值的函数时（如UUID()或RAND()）

下列示例查询使用了用户变量（@status），此情形下，WHERE子句中查询的select_type为UNCACHEABLE SUBQUERY。

```
EXPLAIN
SELECT *
FROM employees e
WHERE e.emp_no = (
        SELECT @status FROM dept_emp de WHERE de.dept_no='d005'
);
```

id	select_type	table	type	key	key_len	ref	rows	Extra
1	PRIMARY	e	const	PRIMARY	4	const	1	Using where
2	UNCACHEABLE SUBQUERY	de	ref	PRIMARY	12	const	1	Using where; Using index

10. UNCACHEABLE UNION

前面已经讲过很多关于UNION与UNCACHEABLE的内容，它们的基本含义很容易理解。UNCACHEABLE UNION是由UNCACHEABLE与UNION两个关键字的属性混合而成的select_type。UNCACHEABLE UNION在MariaDB 5.0中不存在，是从MariaDB 5.1开始新增的一种select_type。

11. MATERIALIZED

MATERIALIZED是从MariaDB 5.3与MySQL 5.6开始引入的一种select_type，主要用于优化FROM子句或IN(subquery)查询中的子查询。下列查询用于获取所有薪资大于100而小于1000的雇员的信息。

```
EXPLAIN
SELECT *
FROM employees e
WHERE e.emp_no IN (SELECT emp_no FROM salaries WHERE salary BETWEEN 100 AND 1000);
```

MySQL 5.6 或 MariaDB 5.2 版本中，执行上述查询时会先读取 employees 数据表，然后针对 employees 数据表的每条记录执行读取 salaries 数据表的子查询。但从 MariaDB 5.3 开始，处理时会先进行优化处理，即子查询内容具体化（Materialization）为临时表，然后再将临时表与 employees 数据表连接起来。下面是上述示例查询的执行计划。从表中可以看到，子查询部分（执行计划中 id=2 的记录）首先得到处理，即具体化为临时表，其 select_type 标记为 MATERIALIZED。

id	select_type	table	type	Key	key_len	Ref	rows	Extra
1	PRIMARY	<subquery2>	ALL	NULL	NULL	NULL	1	
1	PRIMARY	e	eq_ref	PRIMARY	4	salaries.emp_no	1	
2	MATERIALIZED	salaries	range	ix_salary	4	NULL	1	Using where; Using index

关于对 IN(subquery) 与 FROM 子句中子查询的优化，后面讲解优化器开关选项时会详细说明。此处只要知道 select_type 中的 MATERIALIZED 关键字功能与 DERIVED 类似，用于将查询内容生成为临时表即可。

12. INSERT

从 MariaDB 10.0 开始，用户也可以查看 INSERT、UPDATE、DELETE 语句的执行计划。MariaDB 5.5 与之前的版本中，要想查看 UPDATE、DELETE 语句的执行计划，需要先将 UPDATE 语句变为 SELECT 语句，但这与实际还是有一些距离的。并且，如果 UPDATE、DELETE 语句中使用了连接，那就很难使用上面这个方法。

INSERT 语句的执行计划中，只有 select_type 标记为 INSERT，其他显示部分与 SELECT 查询语句类似。

```
EXPLAIN
INSERT INTO employees VALUES (1, '2014-01-01', 'Matt', 'Lee', 'M', '2014-01-02');
```

执行简单的 INSERT 语句时，几乎没有必要查看有关数据表的访问方法或索引使用等内容。

id	select_type	table	type	key	key_len	ref	rows	Extra
1	INSERT	employees	ALL	NULL	NULL	NULL	NULL	NULL

但查看类似 INSERT INTO … SELECT …这种略复杂语句的执行计划时，select_type 中显示的不再是 INSERT 关键字，而是 SIMPLE 等关键字，与访问 SELECT 查询的执行计划一样。下列执行计划将 employees 数据表与 departments 数据表连接后的结果插入（INSERT）dep_emp 数据表。

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	departments	index	PRIMARY	12	NULL	9	Using index
1	SIMPLE	employees	index	ix_hiredate	3	NULL	300024	Using index; Using join buffer (flat, BNL join)

上面虽然是INSERT语句的执行计划，但显示在select_type域中的并不是INSERT关键字，而是SIMPLE关键字，与查看SELECT查询的执行计划时显示的一样。如果是UPDATE语句或DELETE语句，又会显示什么呢？下面看一个简单UPDATE语句的执行计划。

```
EXPLAIN
UPDATE employees SET gender='F' WHERE first_name='Matt';
```

执行UPDATE语句时，由于要实际查找需要更新的目标记录，所以显示出的信息会更详细，比如如何读取以及读取哪些索引等。但显示在select_type中的不是UPDATE，而是SIMPLE。

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	employees	range	ix_firstname	44	NULL	233	Using where

其实，无论select_type是否为UPDATE都不怎么重要。只要可以查看INSERT、UPDATE、DELETE语句的执行计划，这本身就是非常有用的。

4.3.3 table列

MariaDB的执行计划在显示时不以单位SELECT查询为基准，而以数据表为基准。若数据表有别名，则使用数据表别名。像下面这样，对于不涉及数据表的SELECT查询，执行计划的table列显示为NULL。

```
EXPLAIN SELECT NOW( );
EXPLAIN SELECT NOW( ) FROM DUAL;
```

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	NULL						No tables used

提示 一部分DBMS要求，使用SELECT语句时必须带有FROM子句。为此，经常使用拥有标量值的DUAL数据表（只有一行一列）。而MariaDB中，即使不用FROM子句，查询也能正常执行。但为了与其他DBMS保持兼容，在MariaDB中也可以使用FROM DUAL，而不会引发什么问题。其实，MariaDB服务器中并不真实存在DUAL，但是MariaDB服务器能够很好地进行处理，就像存在DUAL数据表一样。上述示例中，执行第二个带FROM DUAL的查询不会有任何问题，它与第一个查询是一样的。

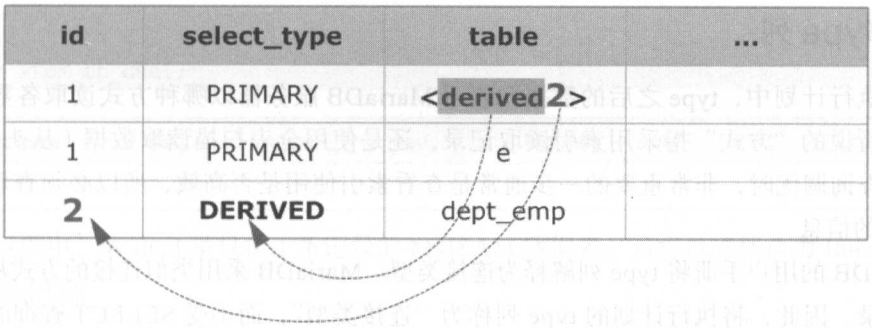
Table列中经常出现类似<derived N>或<union M, N>等由<>包围起来的名称，它表示该表为临时表。并且<>中经常带有数字，它是单位SELECT查询的id。请看下列执行计划。

id	select_type	table	type	Key	key_len	ref	rows	Extra
1	PRIMARY	<derived2>	ALL				10420	
1	PRIMARY	e	eq_ref	PRIMARY	4	de1.emp_no	1	
2	DERIVED	dept_emp	range	ix_fromdate	3		20550	

上述执行计划表格中，第一行的 table 列值为 <derived2>，表示是由 id 为 2 的单位 SELECT 查询的执行计划生成的派生表。也就是说，派生表中存储着 id 为 2 的 SELECT 查询从 dept_emp 数据表获取的结果。图 4-6 表示执行计划的 table 列信息解析方法。

我们至此一起学习了执行计划表中的 id 列、select_type 列、table 列。这三列是判断执行计划中各行数据表执行顺序的依据。下面通过这三列分析上述执行计划。

- 第一行的数据表名为 <derived2>，它表示 id 为 2 的查询先执行，查询结果保存到派生表。
- 从第三行 id 为 2 的查询来看，select_type 列值为 DERIVED，它读取 dept_emp 数据表（显示到 table 列），并创建派生表。
- 第三行分析完成后，再次返回执行计划的第一行。



id	select_type	table	...
1	PRIMARY	<derived2>	
1	PRIMARY	e	
2	DERIVED	dept_emp	

图 4-6 <derived N> 的含义

- 执行计划表中，第一行与第二行拥有相同的 id 值。由此可知，查询需要将两个数据表（第一行中的 <derived2> 与第二行中的 e 数据表）连接起来。但由于 <derived2> 数据表显示在 e 数据表之上，所以 <derived2> 是驱动表，e 是被驱动表。也就是说，先读取 <derived2> 数据表，再连接到 e 数据表。

现在大家对 MariaDB 读取查询执行计划的顺序有了大致了解，刚才分析的执行计划的实际查询如下，供各位参考。

```
SELECT *
FROM
  (SELECT de.emp_no FROM dept_emp de GROUP BY de.emp_no) tb,
  employees e
WHERE e.emp_no=tb.emp_no;
```

从 MariaDB 10.0 开始，子查询的优化有了很大改进，前面讲解 select_type 域值时已提及 MATERIALIZED。在 select_type 为 MATERIALIZED 的执行计划中，table 列会出现类似 <subquery N> 的值。这表示将子查询结果具体化（Materialization）为临时表，其实可以将其理解为与 <derived N> 类似的方法。

提示 与其他 DBMS 不同，MariaDB 要求 FROM 子句中的子查询（Derived，派生表）必须带有别名。否则会显示没有别名的错误信息，并终止执行查询。编写查询或查看执行计划时，请一定要记得写上别名。

```
MariaDB> SELECT dttm FROM (SELECT NOW( ) AS dttm);
ERROR 1248 (42000): Every derived table must have its own alias

MariaDB> SELECT dttm FROM (SELECT NOW( ) AS dttm) derived_table_alias;
+---------------------+
| datetime            |
+---------------------+
| 2011-02-05 14:57:23 |
+---------------------+
```

4.3.4 type 列

查询执行计划中，type 之后的数据列表示 MariaDB 服务器以哪种方式读取各数据表的记录。此处所说的“方式”指采用索引读取记录，还是使用全表扫描读取数据（从头到尾读取）等。进行查询调优时，非常重要的一步通常是查看索引使用是否高效，所以必须查看执行计划中 type 列的信息。

MariaDB 的用户手册将 type 列解释为连接类型。MariaDB 采用类似连接的方式从一个数据表读取记录。因此，将执行计划的 type 列称为“连接类型”，而不受 SELECT 查询的数据表个数影响。但请不要将其与连接进行过多联系，理解为各数据表的访问方式（Access type）即可。

执行计划的 type 列有如下 12 个值，这些值在当前使用的大部分版本（MySQL 5.0、5.1、5.6 及 MariaDB 5.x、MariaDB 10.0）中几乎没什么差别。

(1) system
(2) const
(3) eq_ref
(4) ref
(5) fulltext
(6) ref_or_null
(7) unique_subquery
(8) index_subquery
(9) range
(10) index_merge
(11) index
(12) ALL

上述 12 种访问方法中，除最下方的 ALL 之外，其他所有访问方法都使用索引。ALL 访问方法不用索引，它从数据表读取记录时采用全表扫描的方式，即从头到尾读取数据记录。一个单位 SELECT 查询只能使用其中一种访问方法。此外，以上访问方法中，除 index_merge 外，其

他方法都只能使用一种索引。因此，执行计划的各行不会出现 2 种以上的访问方法，index_merge 之外的 type 中，索引项也只显示一种索引名。

下面依次介绍执行计划的 type 列值，请注意，该顺序同时也是其性能快慢的排序（MariaDB 赋予的优先级），对各 type 的介绍也将按照此顺序进行。MariaDB 优化器会同时分析计算访问方式及其代价，并从中选择一个代价最小的访问方式。

1. system

访问仅有 1 条记录的数据表或没有记录的空表时，使用的访问方法称为“system”。该访问方法只用于 MyISAM 或 MEMORY 数据表，而不会用于使用 InnoDB 或 XtraDB 存储引擎的数据表。

```
EXPLAIN
SELECT * FROM tb_dual;
```

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	tb_dual	system				1	

上面示例中，tb_dual 是只含 1 条记录的 MyISAM 数据表。若将该表转换为 InnoDB 表，会有怎样的结果呢？

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	tb_dual	index	PRIMARY	1		1	Using index

虽然结果可能会根据查询形式的不同而有所不同，但访问方法（type 列）通常都是 ALL 或 index。数据表中只有 1 条记录或没有记录时，会使用 system 访问方法。因此，实际应用程序所用的查询中几乎看不到这种执行计划。

2. const

在不受数据表记录条数限制的前提下，查询中含有使用主键或唯一键列的 WHERE 条件子句时，通常采用 const 方式处理这种查询，查询一定会返回 1 行记录。其他 DBMS 也将其称为唯一索引扫描（UNIQUE INDEX SCAN）。

```
EXPLAIN
SELECT * FROM employees WHERE emp_no=10001;
```

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	employees	const	PRIMARY	4	const	1	

如下列示例所示，多个列组成的主键或唯一键中，仅使用索引的部分列作为条件时，无法使用 const 类型的访问方法。此时，即使实际只存储 1 条记录，MariaDB 引擎也不会读取数据，因而也就无法确认记录为 1 条。

```
EXPLAIN
SELECT * FROM dept_emp WHERE dept_no='d005';
```

只将主键的一部分用作条件时，访问方式将为 ref 而非 const。

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	dept_emp	ref	PRIMARY	12	const	53288	Using where

但若在 WHERE 子句中将主键或唯一索引的所有列明确作为同等条件，则使用 const 访问方法，如下所示。

```
EXPLAIN
SELECT * FROM dept_emp WHERE dept_no='d005' AND emp_no=10001;
```

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	dept_emp	const	PRIMARY	16	const,const	1	

提示 type 列为 const 的执行计划中，MariaDB 优化器在查询优化过程中会将整个查询常量化，所以执行计划的 type 列值标记为“常量”（const）。请看下列查询示例。

```
EXPLAIN
SELECT COUNT(*)
FROM employees e1
WHERE first_name=(SELECT first_name FROM employees e2 WHERE emp_no=100001);
```

上述示例查询中，WHERE 子句含有子查询，用于检索 employees（e2）数据表的主键，并读取 first_name。该查询的执行计划如下，与预想的一样，e2 数据表使用 const 类型访问主键。

id	select_type	table	type	key	key_len	ref	rows	Extra
1	PRIMARY	e1	ref	ix_firstname	44		247	Using where
2	SUBQUERY	e2	const	PRIMARY	4	const	1	

该查询经过优化器优化后转换为如下查询。优化时，优化器先将子查询常量化，然后传递给查询执行器，故访问方式为 const。

```
SELECT COUNT(*)
FROM employees e1
WHERE first_name='Jasminko'; --// Jasminko 是员工号为 100001 雇员的 first_name 值
```

3. eq_ref

eq_ref 访问方法只出现在要连接多个表的查询执行计划中。在连接中将先读取的数据表列

值用作接下来要读取的数据表的主键或唯一键列的检索条件时，这种方法即为 eq_ref。此时，从第二个以后读取的数据表的 type 列都显示为 ea_ref。并且，用唯一键检索第二个以后读取的数据表时，唯一键索引必须为 NOT NULL。若主键或唯一索引由多个数据列组成，只有索引的全部列用作比较条件，eq_ref 访问方法才可用。也就是说，只有保证连接中第二个以后读取的数据表中必有 1 条记录，才可以使用 eq_ref 访问方法。

请看下列查询的执行计划。从执行计划表格中可以看到，第一行与第二行的 id 都为 1，所以会执行两个数据表的连接操作。并且，由于 dept_emp 数据表位于执行计划上方，所以先读取 dept_emp 数据表，然后使用“e.emp_no=de.emp_no”条件检索 employees 数据表。employees 数据表的 emp_no 是主键，所以执行计划的第二行的 type 列显示为 eq_ref。

```
EXPLAIN
SELECT * FROM dept_emp de, employees e
WHERE e.emp_no=de.emp_no AND de.dept_no='d005';
```

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	de	ref	PRIMARY	12	const	1	Using where
1	SIMPLE	e	eq_ref	PRIMARY	4	employees.de.emp_no	1	

4. ref

与 ea_ref 不同，ref 访问方法在使用时与连接顺序无关，也没有主键或唯一键等约束条件。使用相等（Equal）条件检索时，使用 ref 访问方法不受索引类型影响。ref 类型无法保证必有 1 条记录返回，所以不比 const 或 eq_ref 快。但由于只对相等条件进行比较，也可以说是一种非常快的访问方法。

```
EXPLAIN
SELECT * FROM dept_emp WHERE dept_no='d005';
```

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	dept_emp	ref	PRIMARY	12	const	53288	Using where

从上述示例可知，构成 dep_emp 数据表主键的数据列（dept_no+emp_no）中，由于 WHERE 子句只包含部分相等条件，所以无法保证必有 1 条符合条件的记录。使用 ref 访问方法（非 const）时，执行计划的 ref 列值标记为 const。该 const 不是访问方法，是指用作 ref 比较方式的输入常数值（'d005'）。后面会再次介绍有关 ref 列的内容。

下面简单总结一下前面学过的有关执行计划的 type 内容。

const

不受连接顺序影响，对主键或唯一键的所有列使用相等条件进行检索（返回且只返回 1 条记录）。

eq_req

使用连接中第一次读取的数据表列值，将第二张数据表用作主键或唯一键进行相等条件检索（第二张

数据表返回且只返回 1 条记录）。

ref

不受连接顺序和索引类型限制，使用相等条件进行检索（允许不保证只返回 1 条记录）。

上述三种访问方法的共同点是，它们的 WHERE 子句中使用的比较运算符必须是相等比较运算符。相等比较运算符指“=”或“<=>”，“<=>”只在比较 NULL 值时与“=”不同，其他情形与“=”相同。关于“<=>”运算符的更多内容将在第 7 章进行详细讲解。

以上三种方法都是非常好的访问方法，只要索引的分布图符合要求，使用它们都不会引发性能问题，查询调优时不必在这三种访问方法上过分纠结。

5. fulltext

fulltext 是使用 MariaDB 全文检索（Fulltext）索引读取记录的访问方法。一般而言，当前查看的 type 的顺序即是处理性能的顺序，但实际根据数据的分布或记录条数，快慢顺序会变得不同。这也是在基于代价的优化器中使用统计信息计算代价的原因。但全文检索索引不管理统计信息，若想使用全文检索索引，则要使用完全不同的 SQL 语法。因此，执行全文检索查询（..MATCH..AGAINST 语句）时，MariaDB 优化器不会考虑代价，几乎每次都会使用 fulltext 访问方法。当然，使用比 fulltext 快的 const、eq_ref 或 ref 访问方法的查询中，不必非得使用 fulltext 访问方法。

MariaDB 的全文检索条件优先级相当高。若用全文索引的条件与其他普通索引的条件在查询中同时使用，普通索引的访问方法不是 const、eq_ref、ref 时，MariaDB 一般会选择使用全文索引的条件进行处理。

全文检索使用“MATCH...AGAINST...”语句执行，相应数据表中必须备有用于全文检索的索引。若数据表中无全文索引，查询就会发生错误而终止。请看下面“MATCH...AGAINST...”示例查询。

```
EXPLAIN
SELECT *
FROM employee_name
WHERE emp_no=10001
  AND emp_no BETWEEN 10001 AND 10005
AND MATCH(first_name, last_name) AGAINST('Facello' IN BOOLEAN MODE);
```

上述查询语句带有三个条件。第一个条件是只访问 employee_name 数据表的一个主键的 const 类型的条件，第二个条件是下面要说明的 range 类型的条件，第三个条件是全文检索条件。该语句的执行计划如下所示。

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	employee_name	const	PRIMARY	4	const	1	Using where

MariaDB 优化器最终选择的是第一个条件，即 const 类型条件。若 const 类型的第一个条件

不存在，那选择其余二者中的哪一个呢？下列执行计划是删除第一个条件后的执行计划。

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	employee_name	fulltext	fx_name	0		1	Using where

从上述执行计划可以看到，并未选择 range 类型的第二个条件，而选择了第三个条件，即全文检索条件。若查询中使用全文检索条件（MATCH…AGGAINST…），MariaDB 通常会毫不犹豫地选择 fulltext 访问方式。但从以往的经验看，使用普通索引的 range 访问方法通常要比使用全文搜索索引的 fulltext 快得多。使用全文检索查询时，最好先查看各条件的执行性能。

提示 到 MySQL 5.5 与 MariaDB 5.5 时，只针对使用 MyISAM 存储引擎的数据表才能生成全文检索引擎。但从 MariaDB 10.0 与 MySQL 5.6 开始，对使用 InnoDB 存储引擎的数据表也可以使用全文搜索引擎。但令人遗憾的是，MariaDB 10.0 与 MySQL 5.6 只支持基于分隔符的索引，即以 B-Tree 形式管理空格、句子符号等分隔的单词，这意味着无法检索单词的局部（特别是单词的后半部）。

6. ref_or_null

该访问方法与 ref 访问方法相似，但增加了 NULL 比较。顾名思义，ref_or_null 是 ref 方式或 NULL 比较（IS NULL）方式，实际业务中用得并不多，大致了解即可。

```
EXPLAIN
SELECT * FROM titles WHERE to_date='1985-03-01' OR to_date IS NULL;
```

id	select_type	table	Type	key	key_len	ref	rows	Extra
1	SIMPLE	titles	ref_or_null	ix_todate	4	const	2	Using where; Using index

7. unique_subquery

unique_subquery 访问方式用于 WHERE 条件中 IN(subquery) 形式的查询。子查询只返回不重复的唯一值时，使用该访问方法。

```
EXPLAIN
SELECT * FROM departments WHERE dept_no IN (
  SELECT dept_no FROM dept_emp WHERE emp_no=10001);
```

id	select_type	table	type	key	key_len	ref	rows	Extra
1	PRIMARY	departments	index	ux_deptname	123		9	Using where; Using index
2	DEPENDENT SUBQUERY	dept_emp	unique_subquery	PRIMARY	16	func,const	1	Using index; Using where

请看上述查询语句中 IN(subquery) 部分的 subquery。由于 emp_no=10001 的记录中，部门号不重复（dept_emp 数据表中主键为 dept_no+emp_no），所以执行计划第二行中 dept_emp 数据表

的访问方法显示为 unique_subquery。

注意 讲解 unique_subquery 与 index_subquery 时，所用的执行计划示例是 MySQL 5.1 中给出的执行计划。MySQL 5.6 中，IN(subquery) 形式的查询处理方式得到一定改善，MariaDB 10.0 中改善更多。因此，MariaDB 10.0 与 MySQL 5.6 中不会再出现类似 MySQL 5.1 中的不合理的处理方式。unique_subquery 与 index_subquery 在 IN(subquery) 形式的查询执行计划中经常出现，但以后各位查看执行计划时将不会经常看到它们。

讲解 unique_subquery 与 index_subquery 时，用到的查询在 MariaDB 10.0 与 MySQL 5.6 中得到改善，优化器将使用连接执行查询。

8. index_subquery

IN 运算符的特点是，对于 IN(subquery) 或 IN（常数列表）形式的条件，先将括号中值列表的重复值删除。前面所学的 unique_subquery 访问方法中，由于它可以保证 IN(subquery) 条件的 subquery 不会产生重复值，所以不需要删除重复值操作。但实际业务中，IN(subquery) 的 subquery 有可能返回重复值，此时可以使用索引删除子查询结果的重复值，所用的访问方法即为 index_subquery。

下面比较 index_subquery 与 unique_subquery 访问方法，以帮助各位更准确地进行理解。

unique_subquery

IN(subquery) 条件的 subquery 不会返回重复值，所以不需要进行删除重复值的操作。

index_subquery

IN(subquery) 条件的 subquery 可能返回重复值，但使用索引可以删除重复值。

其实，index_subquery 与 unique_subquery 都可以用非常低的代价删除 IN() 中的重复值。

下列查询中，IN 运算符内的子查询使用 dept_no 检索 dept_emp 数据表。dept_emp 数据表的主键为 (dept_no+emp_no)，所以子查询读取列值为 d001~d003 的 dept_no 数据列（主键），只获取 dept_no 值。此外，由于主键已经依据 dept_no 列值进行排序，所以不需要为删除重复的 dept_no 另外进行排序。

```
EXPLAIN
SELECT * FROM departments WHERE dept_no IN (
  SELECT dept_no FROM dept_emp WHERE dept_no BETWEEN 'd001' AND 'd003');
```

id	select_type	table	type	key	key_len	ref	rows	Extra
1	PRIMARY	departments	index	ux_deptname	122		9	Using where; Using index
2	DEPENDENT SUBQUERY	dept_emp	index_subquery	PRIMARY	12	func	18626	Using index; Using where

9. range

range 就是我们熟知的索引范围扫描所用的访问方法。range 表示不是依据单个值，而是在一定范围内检索索引。检索索引时主要使用“<、>、IS NULL、BETWEEN、IN、LIKE”等运算符。一般而言，range 访问方法应用程序的查询最常用。从本书介绍的访问方法的顺序看，它在 MariaDB 的所有访问方法中的优先级相当低。但该访问方法速度相当快，所有查询即使只使用该方法也能获得较好的访问性能。

```
EXPLAIN
SELECT * FROM employees WHERE emp_no BETWEEN 10002 AND 10004;
```

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	employees	range	PRIMARY	4	NULL	3	Using where

注意 请注意，本书提及的索引范围扫描是 const、ref、range 三种访问方法的统称。此外，“高效使用索引”与“在范围受限条件下使用索引”这两种表达方式也包含这三种访问方法。与开发者或 DBA 沟通时，也几乎不将 const、ref、range 三种访问方法分开说，一般会说“索引范围扫描”或“范围扫描”。

10. index_merge

与前面介绍的访问方式不同，index_merge 访问方式中，先使用两个以上的索引分别获得搜索结果，然后再将其合并。但从目前的经验看，该访问方式并不像其名称那样高效。index_merge 访问方式有如下几个特点：

- 由于要读取多个索引，所以通常比 range 访问方式的效率差。
- AND 与 OR 连接成的复杂查询中，很多时候不能正常优化。
- 使用全文检索索引的查询中，不使用 index_merge。
- 使用 index_merge 访问方式处理的结果总是两个以上的集合，所以需要对两个集合进行交合并、删除重复等额外操作。

MariaDB 使用手册中，index_merge 访问方法的优先级仅次于 ref_or_null。但本书基于以上几个原因，将其优先级顺序排在 range 访问方式之下。使用 index_merge 访问方式时，执行计划中还要有一些附件信息，这部分内容将在执行计划的 Extra 章节进行详细介绍。

下列查询的 WHERE 子句中，两个条件通过 OR 连接起来，并且两个由 OR 连接的条件分别使用了不同索引。因此，MariaDB 优化器创建的执行计划是：对于“emp_no BETWEEN 1001 AND 11000”条件使用 employees 数据表的主键进行查询，对于“first_name='Smith'”条件使用 ix_firstname 索引进行查询，然后再将两个结果进行合并。

```
EXPLAIN
SELECT * FROM employees
WHERE emp_no BETWEEN 10001 AND 11000
OR first_name='Smith';
```

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	employees	index_merge	PRIMARY, ix_firstname	4,44		1521	Using union (PRIMARY,ix_firstname); Using where

11. index

许多人对 index 访问方法有误解。由于该访问方法的名称为 index，很多不熟悉 MariaDB 的用户认为它是一种高效使用索引的方法。其实 index 访问方式是一种索引全扫描方法，它会从头到尾读取索引。index 访问方式不像 range 访问方式那样只读取需要的索引，因而不如 range 访问方式高效。

与全表扫描方式（从头到尾读取整个表）相比，index 访问方式也要比较同等记录数。但索引文件通常要比数据文件小，因而比全表扫描高效，处理速度也更快。此外，在 index 访问方式下也可以依据查询内容使用已经排好序的索引，从而获得比全表扫描更快的处理速度。下列条件中，若某个查询满足条件 1+ 条件 2 或者满足条件 1+ 条件 3，则使用 index 访问方法。

(1) 无法通过 range、const 或 ref 访问方式使用索引。

(2) 仅用 index 中的列即可处理的查询（即不需要读取数据文件）。

(3) 可以使用索引进行排序或编组的情形（即不需要另外排序）。

下列查询由于不含任何 WHERE 条件，所以无法使用 range、const 或 ref 访问方式。但因为要排序的列有索引（ux_deptname），所以使用 index 访问方式避免另外的排序处理。

```
EXPLAIN
SELECT * FROM departments ORDER BY dept_name DESC LIMIT 10;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	departments	index		ux_deptname	123		9	Using index

上述示例的执行计划虽然使用 index 访问方式，从头到尾读取数据表的索引，但由于存在 LIMIT 条件，只要逆序读取 10 个索引即可，所以效率还是相当高的。如果没有 LIMIT 条件限制或要读取的记录非常多，index 访问方式就会变得相当慢。

12. ALL

ALL 就是我们常说的全表扫描方式，该方式从头到尾读取数据表中的全部记录，然后删除不需要的记录（存在检测条件时），再返回符合要求的记录。全表扫描是一种非常低效的方式，前面介绍的所有访问方法不可用时才选用该方法。

与其他 DBMS 一样，InnoDB 也提供了一次读取多页的功能。进行全表扫描或索引全扫描这类需要大量磁盘 I/O 的作业时，它相当有用。InnoDB 中，该功能称为“预读”（Read Ahead），使用该功能可以一次读取并处理多个页。像数据仓库、批处理程序一样，需要处理大量记录的查询中，ALL 访问方法可能比其他调优失败的查询（查询调优时强制使用索引）更好。请注意，

查询调优并不意味着无条件禁用索引全扫描或全表扫描。

由于 index 与 ALL 访问方法不限制扫描范围，所以一般不适合应用于 OLTP（On-Line Transaction Processing，联机事务处理）环境，比如需要立即将响应发送给用户的 Web 服务等。若数据表不是很小，可以先将一定量的数据存储到数据表，然后使用该访问方式查看查询的性能。

提示 MariaDB 中若连续发生读取相邻页面，则后台读取线程会从磁盘一次读取 64 个页面（最大页面数），这比每次读取一个页面的方式要快得多。这种预先读取的方式就称为“预读”。

4.3.5 possible_keys 列

执行计划中的 possible_keys 列也经常会引起用户的误解。处理查询时，MariaDB 优化器会先考虑多种处理方法，然后从中选择一个代价最低的执行计划执行查询。MariaDB 优化器为了创建最优执行计划，在选定的访问方式中使用候选索引列表，这就是 possible_keys 列中的内容，即有可能使用的索引目录。从实际执行计划看，数据表的所有索引通常都会出现在该目录中，这对查询调优没什么用处。因此，查看执行计划时可以忽略 possible_keys 列。请注意，possible_keys 列中列出的索引并非都会用到。

4.3.6 key 列

如前所述，possible_keys 列中的索引是候选索引，而 key 列中显示的索引则是最终选用的执行计划所使用的索引。因此，进行查询调优时，一定要查看 key 列中显示的索引是否是想要的索引。key 列中显示的值为 PRIMARY 时，表示使用主键，除此之外的其他值均为创建数据表或索引时赋予的固有名称。

执行计划的 type 列值不是 index_merge 时，每个数据表只能使用 1 个索引。但使用 index_merge 执行计划时，则可以使用 2 个以上的索引。此时 key 列将显示多个索引，它们之间以逗号（,）进行分隔。再回顾一下前面的 index_merge 执行计划，从下列执行计划可以看到，WHERE 子句中的各个条件使用 PRIMARY 与 ix_firstname 索引。

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	employees	index_merge	PRIMARY,ix_firstname	4,44		1521	…

另外，执行计划的 type 为 ALL 时，即完全不使用索引时，key 列将显示为 NULL。

提示 MariaDB 中，主键默认名为 PRIMARY，不能赋予其他名称。除此之外，其他索引皆在创建数据表或索引时赋名。使用执行计划或查询提示时，若要指称主键，则可以直接使用 PRIMARY 关键字。

4.3.7 key_len 列

虽然 key_len 列包含非常重要的信息，但许多用户经常忽略它。实际业务中使用的数据表更

多采用多列索引，而不是单列索引。执行计划的 key_len 列值会告诉我们用于处理查询的多列索引是由几个数据列构成的，更准确地说，是告诉我们索引的各个记录用到了几字节。因此，无论是在多列索引还是单列索引中，它都提供了相同的指标。

下面示例用于查询 dept_emp 数据表，该数据表的主键由两个数据表构成（dept_no+ emp_no）。该示例查询在 dept_emp 数据表的主键中，只将 dept_no 用于比较。

```
EXPLAIN
SELECT * FROM dept_emp WHERE dept_no='d005';
```

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	dept_emp	ref	PRIMARY	12	const	53288	Using where

因此，key_len 列值为 12。也就是说，由于 dept_no 数据列的类型为 CHAR(4)，所以在主键中只能有效使用前面 12 字节。该数据表的 dept_no 列使用 utf8 字符集，而一个 utf8 字符占用的空间是可变的，约 1~3 字节。MariaDB 服务器为 utf8 字符分配内存空间时，按固定的 3 字节计算，不受字符影响。因此，上述执行计划中，key_len 列值为 12 字节（4*3 字节）。

下列查询使用与前面相同的索引，但在两个条件中分别使用了 dept_no 列与 emp_no 列。

```
EXPLAIN
SELECT * FROM dept_emp WHERE dept_no='d005' AND emp_no=10001;
```

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	dept_emp	const	PRIMARY	16	const,const	1	

dept_emp 数据表的 emp_no 列的类型为 INTEGER，INTEGER 类型占用 4 字节。上述查询语句的条件中不仅用到了 dept_no 列，也用到了 emp_no 列，它们都是数据表的主键。因此，key_len 列值应为 dept_no 列长与 emp_no 列长之和，即 16 字节。

此外，有时 key_len 字段值要比数据类型长一些。如下所示，创建数据表，查看查询执行计划。

```
EXPLAIN
SELECT * FROM titles WHERE to_date<='1985-10-10';
```

上述查询的执行计划如下所示。从执行计划表格中可以看到，key_len 列值为 4，ix_todate 索引仅拥有一个 to_date 列，且列类型为 DATE 数据类型。由于 MariaDB 中的 DATE 类型占用 3 字节，所以 key_len 列值应该显示为 3，但为什么显示的是 4 呢?

id	select_type	table	Type	key	key_len	Ref	Rows	Extra
1	SIMPLE	titles	range	ix_todate	4	NULL	51	Using where; Using index

原因在于，to_date 列使用 DATE 类型的同时定义了一个 NULLABLE 列，用于保存 NULL。数据列为 NOT NULL 列时，MariaDB 额外使用 1 字节存储列值是否为 NULL。因此，上述示例

中，key_len 显示的值为 3+1 字节。

还有一点需要注意，对于 key_len 的值，每个版本的 MariaDB 采用不同的基准进行显示。请分别在 MySQL 5.0.x 与 MariaDB 5.5.x 中查看下列查询的执行计划，并比较异同。

```
EXPLAIN
SELECT * FROM dept_emp WHERE dept_no='d005' AND emp_no <> 10001;
```

1. MySQL 5.0 以前版本

查询语句的条件中仍使用了 emp_no 与 dept_no 主键，但是 key_len 值却变为 12。为何从原来的 16 变为 12 了呢？原因在于，key_len 中显示的值仅包含了使用索引限制范围的条件列，并未包含用作检测条件的列。MariaDB 5.0 通过 key_len 列值判断索引中的几字节被用作范围限制条件。

id	select_type	table	Type	key	key_len	Ref	Rows	Extra
1	SIMPLE	dept_emp	ref	PRIMARY	12	const	53298	Using where

2. MariaDB 5.1 以后版本

MariaDB 5.1 中，执行计划的 key_len 显示为 16，而 type 列值由 ref 变为 range。虽然“emp_no<>10001”只是简单的检测条件，但也会一起被包含并计算到 key_len 之中。因此，在 MariaDB 5.1 中无法通过 key_len 列值判断索引中有几字节被用作范围限制条件。

id	select_type	table	type	Key	key_len	ref	rows	Extra
1	SIMPLE	dept_emp	range	PRIMARY	16		53298	Using where

其实，两个版本间的差异是由 MariaDB 引擎与 InnoDB 存储引擎所起作用发生很大变化而引起的。MySQL 5.0 中，只有用作范围限制条件的数据列才会被传递到存储引擎。但是从 MariaDB 5.1 开始，不论范围限制条件还是检测条件，只要用到索引，就会被传递给存储引擎。MariaDB 将这称为“条件下推”（Condition push down），相关内容将在 6.2.10 节详细介绍。

4.3.8 ref 列

访问方法是 ref 时，ref 列显示提供了哪种值作为参考条件（Equal 比较条件）。若指定了常数值，则 ref 列值显示为 const；若为其他数据表的列值，则显示数据表名与列名。对该列中显示的内容也可以不用太费心，但是像下面这个示例就要认真注意了。

有时在查询的执行计划中，ref 列值会显示为 func，它是 Function 的缩写，表示不是原封不动地使用参考值，而是先经过排序变换或数值运算后再引用。请看下列查询的执行计划示例。

```
EXPLAIN
SELECT *
FROM employees e, dept_emp de
WHERE e.emp_no=de.emp_no;
```

上述查询将 employees 数据表与 dept_emp 数据表连接，并未对连接条件中使用的 emp_no 列值做任何变换或加工。因此，该查询的执行计划中，ref 列显示的是连接对象列的名称。

id	select_type	table	type	key	key_len	Ref	rows	Extra
1	SIMPLE	de	ALL				334868	
1	SIMPLE	e	eq_ref	PRIMARY	4	de.emp_no	1	

接下来，在上述查询的连接条件中添加简单的运算表达式，创建新的查询，并查看其执行计划。

```
EXPLAIN
SELECT *
FROM employees e, dept_emp de
WHERE e.emp_no=(de.emp_no-1);
```

在上述查询中先读取 dep_emp 数据表，然后从 de.emp_no 值中减去 1，再用结果值连接 employees。该查询的执行计划中，ref 列值不再是连接列的名称，而显示为“func”。

id	select_type	table	type	key	key_len	Ref	rows	Extra
1	SIMPLE	de	ALL				334868	
1	SIMPLE	e	eq_ref	PRIMARY	4	func	1	Using where

像这样，ref 列不仅在用户明确变换值时显示为 func，而且 MariaDB 服务器要在内部变换值时，ref 列也会显示为 func。最具代表性的是连接两个字符集不同字符串数据列时，或连接数值类型的数据列与字符串类型的数据列时。若有可能，即使 MariaDB 服务器不做这种变换，也要尽可能地使连接数据列的类型保持一致。

4.3.9 rows 列

对于各个查询条件，MariaDB 优化器会列出可能的处理方式，并比较各种处理方式的代价，最终选择一个代价最小的执行计划。计算代价时，要对各种处理方式将读取并比较的记录数进行预测，要调查对象数据表中含有多少记录，以及各个索引值的分布图，以统计信息为基准进行预测。

MariaDB 执行计划的 rows 列用于显示为判断执行计划的效率而预测的记录数。rows 列值是 MariaDB 优化器根据各存储引擎的统计信息推算出的数值，因而不准确。此外，rows 列中显示的数值并不是可能返回的记录条数，而是指处理查询时要从磁盘读取与检查多少条记录。因此，执行计划中的 rows 列值通常与实际查询结果返回的记录条数不一致。

```
EXPLAIN
SELECT * FROM dept_emp WHERE from_date>='1985-01-01';
```

上述示例查询用于从 dept_emp 数据表获取 from_date 大于等于“1985-01-01”的记录。执行示例查询时，虽然可以使用由 dept_emp 数据表的 from_date 列生成的 ix_fromdate 索引，但最

终执行计划选用了全表扫描（ALL）。从下列查询的执行计划中可以看到，rows 列值为 331603，这表示 MariaDB 优化器处理该查询时大约要读取 331603 条记录。dept_emp 数据表的全部记录数为 331603，由此可以判断执行查询时要对数据表的所有记录进行比较。因此，MariaDB 优化器没有选择索引范围扫描，而选择了全表扫描。

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	dept_emp	ALL	ix_fromdate				331603	Using where

接下来更改上述查询语句，进一步缩小查询范围，然后查看其执行计划。从下列查询的执行计划可知，MariaDB 优化器预测只要读取检查大约 292 条记录就可以得到所要的结果。当然，执行计划使用的不再是全表扫描，而是索引范围扫描（range）。

```
EXPLAIN
SELECT * FROM dept_emp WHERE from_date>='2002-07-01';
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	dept_emp	range	ix_fromdate	ix_fromdate	3		292	Using where

上述示例查询中，优化器推测 from_date 列值大于“2012-07-01”的记录只有 292 条，这只占数据表全部记录数的 8.8%。因此，优化器最终采用 range 方式（索引范围扫描）处理 ix_fromdate 索引。此外，由于索引中包含的 from_date 为 DATE 类型，所以 key_len 列显示为 3 字节。

4.3.10 Extra 列

与其名称含义不同，查询执行计划中与性能相关的重要信息经常会显示在 Extra 列。Extra 列中显示的语句只有固定的几个，一般会同时显示 2~3 个。从 MariaDB 5.0 升级到 MariaDB 5.1 后，又新增了几个关键字，而 MariaDB 5.5 与 MariaDB 5.1 差不多。MariaDB 5.1 中新增的关键字都带有类似“（从 MariaDB 5.1 起）”的标记，供各位参考。下面将对可以出现在 Extra 列中的语句逐个分析。请注意，此处的说明顺序与性能优劣无关，各语句的顺序本身没有任何含义。

1. const row not found

查询的执行计划中，虽然使用 const 访问方式读取了数据表，但若实际数据表中没有 1 条符合条件的记录，则 Extra 列将显示“const row not found”信息。Extra 列显示“const row not found”信息时，建议先向数据表存储适当的测试数据，然后再次查看查询的执行计划。

2. Distinct

请看下列示例查询，其执行计划的 Extra 列会显示 Distinct 关键字。

```
EXPLAIN
SELECT DISTINCT d.dept_no
```

```
FROM departments d, dept_emp de WHERE de.dept_no=d.dept_no;
```

id	select_type	table	type	key	key_len	Ref	rows	Extra
1	SIMPLE	d	index	ux_deptname	123	NULL	9	Using index; Using temporary
1	SIMPLE	de	ref	PRIMARY	12	employees.d.dept_no	18603	Using index; Distinct

上述查询中，实际想访问的值为 dept_no，而 dept_no 在 departments 数据表与 dept_emp 数据表中都存在，执行查询时会先将两个数据表连接起来，然后再在结果中进行 DISTINCT 处理。这样，最终得到的结果中，dept_no 就不会有重复。

图 4-7 表示执行计划的 Extra 列为 Distinct 时的查询处理过程。处理查询的 DISTINCT 时，全部忽略不需要连接的记录，只连接必需的记录。因此，只从 dept_emp 数据表中读取必需的记录，如图所示。

3. Full scan on NULL key

Full scan on NULL key 处理经常出现在带有类似 col1 IN(SELECT col2 FROM…) 条件的查询中。若 col1 值为 NULL，则条件最终变为 NULL IN(SELECT col2 FROM…)。SQL 标准将 NULL 定义为“未知值”，也定义了 NULL 的运算规则。根据规则定义，执行运算时需要对条件进行如下比较。

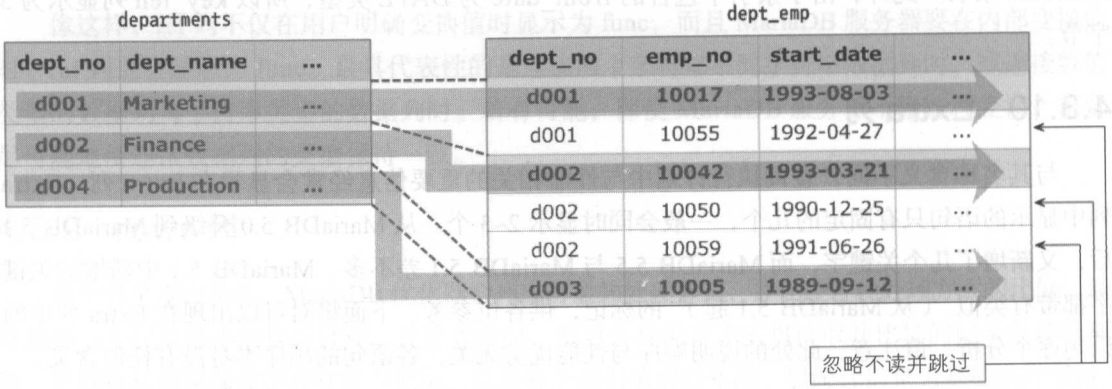


图 4-7 Distinct 的处理方式

- 若子查询拥有至少 1 条结果记录，则最终比较结果为 NULL。
- 若子查询未拥有任何结果记录，则最终比较结果为 FALSE。

比较过程中，若 col1 为 NULL，则必须进行全表扫描才能查询结果。MariaDB 执行查询时，若 col1 为 NULL，则使用全表扫描，Extra 列的 Full scan on NULL key 正用于指明这一点。若 col1 IN(SELECT col2 FROM...) 条件中，col1 列定义为 NOT NULL，则不会使用全表扫描，Extra 列也不会显示 Full scan on NULL key。

请看下列执行计划，其 Extra 列显示为 Full scan on NULL key。

```
EXPLAIN
SELECT d.dept_no, NULL IN (SELECT id.dept_name FROM departments id)
FROM departments d ;
```

id	select_type	table	type	key	key_len	Ref	rows	Extra
1	PRIMARY	d	index	ux_deptname	123	NULL	9	Using index
2	DEPENDENT SUBQUERY	id	index_subquery	ux_deptname	123	const	2	Using index; Full scan on NULL key

虽然数据列未定义为 NOT NULL，但如果也可以忽略这种 NULL 比较规则，那么只要将“col1 绝不可能为 NULL”这一事实告知 MariaDB 优化器即可。最具代表性的方法是，在该查询条件中设置 col1 IS NOT NULL。这样，col1 为 NULL 时，col1 IS NOT NULL 条件就为 FALSE，也就不会执行 col1 IN(SELECT col2 FROM tb_test2) 条件。

```
SELECT *
FROM tb_test1
WHERE col1 IS NOT NULL
        AND col1 IN (SELECT col2 FROM tb_test2);
```

即使执行计划的 Extra 列显示了 Full scan on NULL key，只要 IN 或 NOT IN 运算符左侧的值不为 NULL，就不会进行全表扫描，各位不必担心。然而，IN 或 NOT IN 运算符左侧值为 NULL 时，若子查询中没有特别指定 WHERE 条件，就会引发严重的性能问题。更详细的介绍请参考 7.4.9 节。

注意 请注意，讲解中使用的查询只用于帮助说明相关内容，有时编写的查询可能有些“生硬”，有时为了显示某个特定的执行计划，还会编写一些无实际意义的查询。这些查询只是为了更方便地讲解相关内容，所以请各位不要纠结于示例，不要再浪费时间一个劲地问：“明明有更高效的查询方法，为什么非要这么写呢？”

4. Impossible HAVING

不存在满足查询 HAVING 子句条件的记录时，执行计划的 Extra 列就会显示“Impossible HAVING”关键字。

```
EXPLAIN
SELECT e.emp_no, COUNT(*) AS cnt
FROM employees e
WHERE e.emp_no=10001
GROUP BY e.emp_no
HAVING e.emp_no IS NULL;
```

上述示例的 HAVING 子句虽然含有“e.emp_no IS NULL”条件，但由于 employees 数据表的 e.emp_no 数据列为主键，所以 emp_no 列全为 NOT NULL 类型。因此，绝不可能满足 e.

emp_no IS NULL 条件，所以 Extra 列显示“Impossible HAVING”关键字。

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE								Impossible HAVING

应用程序查询中，若执行计划的 Extra 列显示“Impossible HAVING”信息，大部分表示查询编写错误，此时建议仔细查看编写的查询是否有误。

5. Impossible WHERE

与“Impossible HAVING”类似，WHERE 条件总是为 FALSE 时，Extra 列将显示为“Impossible WHERE”。

```
EXPLAIN
SELECT * FROM employees WHERE emp_no IS NULL;
```

上述查询中，由于 WHERE 条件中用到的 emp_no 列为 NOT NULL，所以 emp_no IS NULL 条件恒为 FALSE。此时，查询的执行计划的 Extra 列将显示“Impossible WHERE”信息。

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE								Impossible WHERE

6. Impossible WHERE noticed after reading const tables

上一小节讲解的“Impossible WHERE”情形下，MariaDB 不会实际读取数据，而是直接通过数据表结构判断条件为“不可能”。那么下列示例查询的 Extra 列会显示什么信息呢？

```
EXPLAIN
SELECT * FROM employees WHERE emp_no=0;
```

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE							Impossible WHERE noticed after reading const tables

如果不实际执行上述查询，就无法判断 emp_no=0 的记录是否存在。然而事实上，优化器只要查看该查询的执行计划就会知道，员工号为 0 的雇员实际并不存在。

由此可知，MariaDB 制定执行计划时会执行查询的一部分。上面说明的查询使用相等条件比较 employees 数据表的主键。通过前面的学习我们已经知道，此时使用 const 访问方式。查询中需要使用 const 访问方式的部分是，制定执行计划阶段优化器会直接执行一部分查询，并将查询结果值替换为原查询中的常数。

```
SELECT *
FROM employees oe
WHERE oe.first_name=(
SELECT ie.first_name
```

```
FROM employees ie
WHERE ie.emp_no=10001
);
```

也就是说，执行上述查询时，对于 WHERE 条件句中的子查询，优化器先用部分执行结果进行如下替换，再正式执行查询操作。

```
SELECT *
FROM employees oe
WHERE oe.first_name='Georgi';
```

7. NO matching min/max row

若所有记录都不满足查询中的 WHERE 条件，Extra 列通常就会显示“Impossible WHERE...”语句。若一个查询中含有类似 MIN() 与 MAX() 的集合函数，且无任何符合查询条件的记录，则 Extra 列将显示“No matching min/max row”信息，并返回 NULL 作为 MIN() 与 MAX() 的结果。

```
EXPLAIN
SELECT MIN(dept_no), MAX(dept_no)
FROM dept_emp WHERE dept_no='';
```

上述查询用于从 dept_emp 数据表检索 dept_no 列为空字符串的记录。由于 dept_no 列全为 NOT NULL，所以没有任何符合条件的记录，查询执行计划的 Extra 列显示出“No matching min/max row”信息。

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE								No matching min/max row

注意 要对 Extra 列显示的信息（如 No matching... 或 Impossible WHERE... 等）有正确认识，否则容易产生误解，误以为查询本身发生错误。Extra 列显示的内容只表示缺少计算执行计划的数据。请注意，不能因为 Extra 列显示这些信息就认为实际查询发生错误。

8. no matching row in const table

请看下列查询，使用 const 方式访问连接中的数据表时，若不存在符合条件的记录，则执行计划的 Extra 列将显示“no matching row in const table”信息。

```
EXPLAIN
SELECT *
FROM dept_emp de,
(SELECT emp_no FROM employees WHERE emp_no=0) tb1
WHERE tb1.emp_no=de.emp_no AND de.dept_no='d005';
```

该信息类型与“Impossible WHERE…”类似，表示缺少制定执行计划的基本数据。

id	select_type	table	type	key	key_len	ref	rows	Extra
1	PRIMARY							Impossible WHERE noticed after reading const tables
2	DERIVED							no matching row in const table

9. NO tables used

不带 FROM 子句的查询或 FROM DUAL 查询的执行计划中，Extra 列会显示“No tables used”信息。与其他 DBMS 不同，MariaDB 允许查询不带 FROM 子句。像这样，查询不带 FROM 子句或 FROM 子句中使用 DUAL（行列均为 1 的虚拟常数表）时，执行计划的 Extra 列就会显示“No tables used”信息。

```
EXPLAIN SELECT 1;
EXPLAIN SELECT 1 FROM dual;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE								No tables used

提示 MariaDB 中，执行不带 FROM 子句的查询不会发生错误。但在 Oracle 中，所有查询必须有引用表，即必须带有 FROM 子句。为了应对不需要 FROM 子句的情形，可以使用 DUAL 常数数据表。此外，MariaDB 为了与 Oracle 保持兼容，也可以在 FROM 子句中明确使用 DUAL 数据表。在 FROM 子句中使用 DUAL 时，MariaDB 优化器在内部会将其视为无 FROM 子句的查询并进行处理。

10. Not exists

应用程序开发过程中会经常使用一些查询，访问一些表 A 中存在但表 B 中不存在的值。此时主要使用 NOT IN(subquery) 或 NOT EXISTS 运算符，这样的连接称为“反连接”（Anti-JOIN）。同样的处理也可以使用左外连接（left OUTER JOIN）实现。通常会用使用 NOT IN(subquery) 或 NOT EXISTS 等运算符的“反连接”进行处理，但记录数很多时，使用外部连接能获得更快的处理性能。

下面介绍使用外部连接的示例查询，它用于访问所有存在于 dept_emp 数据表而不存在于 departments 数据表的 dept_no。下列查询中，先外部连接 departments 数据表，再在 WHERE 子句（非 ON 子句）中检查并获取外部表（departments）中 dept_no 列值为 NULL 的记录。也就是说，反连接方法常常只用于获取普通连接（INNER JOIN）时不会得到的结果。

```
EXPLAIN
SELECT *
FROM dept_emp de
  LEFT JOIN departments d ON de.dept_no=d.dept_no
WHERE d.dept_no IS NULL;
```

像这样使用外部连接进行反连接的查询中，执行计划的 Extra 列会显示 Not exists 信息。请注意，Not exists 信息并不表示将查询变为 NOT EXISTS 形式的查询，MariaDB 会在内部做某种优化，而优化名称即为 Not exists。请不要将 Extra 列的 Not exists 与 SQL 中的 NOT EXISTS 运算符混为一谈。

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	de	ALL				334868	
1	SIMPLE	d	eq_ref	PRIMARY	12	employees.de.dept_no	1	Using where; Not exists

11. Range checked for each record(index map:N)

下列示例查询执行时需要连接两个数据表。连接条件不是常量而是两个变量（e1.emp_no 与 e2.emp_no）时，MariaDB 优化器先读取 e1 数据表，然后为进行连接而读取 e2 时，在索引范围扫描与全表扫描中无法判断哪个更高效。也就是说，每次从 e1 数据表中读取一条记录，e1.emp_no 值就会变化，即用于计算查询代价的基准值一直在变化。因此，优化器无法判断使用哪种访问方法读取 e2 数据表更有效率。

```
EXPLAIN
SELECT *
FROM employees e1, employees e2
WHERE e2.emp_no >= e1.emp_no;
```

比如，假设员工号的范围为 1 ~ 100 000 000，那么执行查询时会从头到尾扫描 e1 数据表，同时从 e2 数据表中查找满足 e2.emp_no>=e1.emp_no 条件的记录。该过程中，e1.emp_no=1 时，要读取 e2 数据表中的所有记录（100 000 000 条）；e1.emp_no=100 000 000 时，只读取 e2 数据表中的 1 条记录，图 4-8 描述了该读取过程。

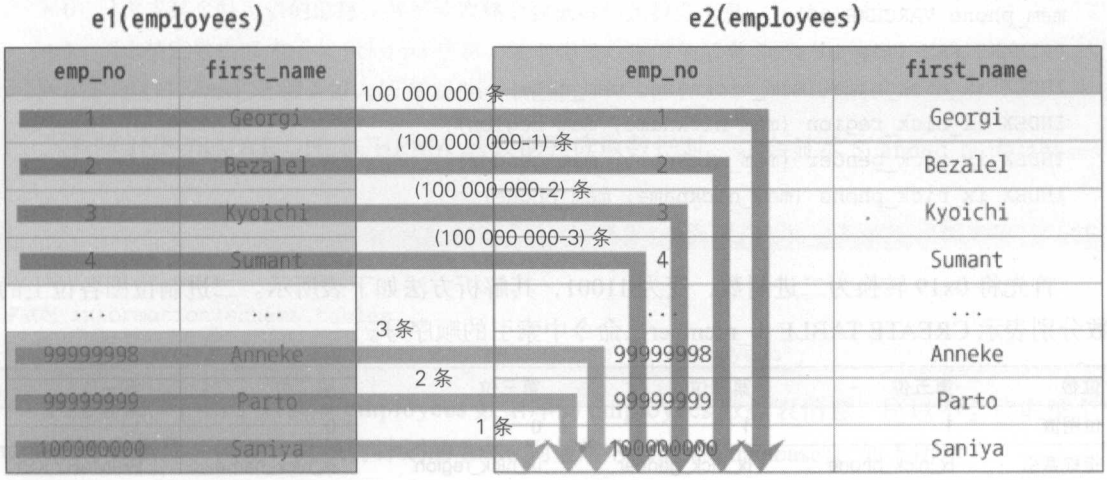


图 4-8 对每条记录都要从 range 与 ALL 中选择

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	e1	ALL		3		300584	Using index
1	SIMPLE	e2	ALL				300584	Range checked for each record (index map: 0x1)

因此，我们采用的优化方法是：e1 数据表的 emp_no 很小时，使用全表扫描访问 e2 数据表；而 e1 数据表的 emp_no 很大时，就使用索引范围扫描访问 e2 数据表。简言之，就是“对每条记录检查索引范围扫描”，此时执行计划的 Extra 列显示为“Range checked for each record”。

Extra 列的显示内容中，“(index map:0x1)”是候选索引号，用于判断是否使用候选索引。“index map”采用 16 进制表示，解析时先要将其变换为二进制数。上述执行计划中，index map 为 0x1，转换为二进制数为 1。因此，处理上述查询时，要根据每条记录确定究竟使用 e2 数据表（employees）的第一个索引还是全表扫描方式。使用 SHOW CREATE TABLE employees 命令访问数据表的结构时，最先显示的索引就是数据表的第一个索引。

此外，查询执行计划的 type 列值为 ALL 时，也表示使用全表扫描进行处理。Extra 列显示为“Range checked for each record”时，type 列会显示为 ALL。也就是说，先检查是否可用“index map”中的候选索引，若候选索引不怎么有用，则最终选用全表扫描，所以 type 列显示为 ALL。

为了帮助各位进一步理解“index map”，下面介绍更复杂的“index map”示例。假设有一个查询，它访问的数据表有多个索引，其执行计划的 Extra 显示“(index map:0x19)”。

```
CREATE TABLE tb_member(
  mem_id INTEGER NOT NULL,
  mem_name VARCHAR(100) NOT NULL,
  mem_nickname VARCHAR(100) NOT NULL,
  mem_region TINYINT,
  mem_gender TINYINT,
  mem_phone VARCHAR(25),
  PRIMARY KEY (mem_id),
  INDEX ix_nick_name (mem_nickname, mem_name),
  INDEX ix_nick_region (mem_nickname, mem_region),
  INDEX ix_nick_gender (mem_nickname, mem_gender),
  INDEX ix_nick_phone (mem_nickname, mem_phone)
);
```

首先将 0x19 转换为二进制数，变为 11001，其解析方法如下表所示。二进制位图各位上的数分别表示 CREATE TABLE tb_member…命令中索引的顺序号。

位数	第五位	第四位	第三位	第二位	第一位
位图值	1	1	0	0	1
指称索引	ix_nick_phone	ix_nick_gender	ix_nick_region	ix_nick_name	PRIMARY KEY

总之，执行计划中，“(index map:0x19)”表示从上表中选择位图值为 1 的位所对应的索引，用作候选索引。

- PRIMARY KEY
- ix_nick_gender
- ix_nick_phone

以每条记录为单位，确定从候选索引中选择使用哪个索引。但是实际使用哪个索引是无法得知的，仅知道各个位图位值为 1 的候选索引及其顺序。

提示 若有大量查询，则这些查询的执行计划的 Extra 列全部显示为“Range checked for each record”。MariaDB 服务器执行这些查询时，使用 SHOW GLOBAL STATUS 命令显示状态值，可以看到 Select_range_check 值会很大。

12. Scanned N databases

从 MariaDB 5.0 开始默认提供 INFORMATION_SCHEMA 数据库。INFORMATION_SCHEMA 数据库集中存放着 MariaDB 服务器内部存在的数据库的元数据信息（数据表、数据列、索引等 Schema 信息）。INFORMATION_SCHEMA 数据库中的所有数据表是读取专用的，仅支持查询。事实上，该数据库内的数据表并不实际包含记录，每当使用 SQL 语句访问时，就会从 MariaDB 服务器的内存中读取并显示元数据信息。这样，一次要访问大量数据表时，就要花费很长时间。

MariaDB 5.1 开始大大提升了访问性能，使用户可以快速访问 NFORMATION_SCHEMA 数据库。通过改善后的查询检索元数据信息时，查询执行计划的 Extra 列会显示 Scanned N databases 信息。Scanned N databases 中，N 表示读取几个数据库的信息，其取值可以为 0、1 或者 all，它们的意义如下。

- 0：只请求某个特定表的信息，并不读取整个数据库的元数据信息。
- 1：请求特定数据库内所有 Schema 信息，读取相关数据库的所有 Schema 信息。
- ALL：读取 MariaDB 服务器内所有 Schema 信息。

只有从 INFORMATION_SCHEMA 的数据表读取数据时，才会显示 Scanned N databases 信息。

```
EXPLAIN
SELECT table_name
FROM information_schema.tables
WHERE table_schema = 'employees' AND table_name = 'employees';
```

上述查询中，由于只读取 employees 数据库的 employees 数据表信息，所以不会读取整个 employees 数据库。因此，其执行计划的 Extra 列显示 Scanned 0 databases，如下所示。

id	select_type	table	Type	..	key	..	..	..	Extra
1	SIMPLE	TABLES	ALL	..	TABLE_SCHEMA, TABLE_NAME	..	..	..	Using where; Skip_open_table; Scanned 0 databases

由于应用程序中几乎不会使用访问 INFORMATION_SCHEMA 数据库中元数据的查询，所以几乎没有执行计划的 Extra 列显示为 Scanned 0 databases 的查询。

13. Select tables optimizer away

查询的 SELECT 语句中只使用 MIN() 或 MAX()，或者用 GROUP BY 访问 MIN() 或 MAX() 时，若无法使用合适的索引，就会按升序或降序只读取 1 个索引。经过这种优化后，执行计划的 Extra 列就会显示 Select tables optimized away 信息。

此外，对于 MyISAM 数据表，SELECT 语句中只有 COUNT(*) 时（不含 GROUP BY）也会采用这种形式的优化。由于 MyISAM 数据表中单独管理全体记录条数，所以即使不读取索引或数据也能快速访问所有记录。但若 WHERE 子句含有条件，则无法使用这种优化。

```
EXPLAIN
SELECT MAX(emp_no), MIN(emp_no) FROM employees;

EXPLAIN
SELECT MAX(from_date), MIN(from_date) FROM salaries WHERE emp_no=10001;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE								Select tables optimized away

第一个查询中，由于 employees 数据表中的 emp_no 列创建有索引，所以可以应用 Select tables optimized away 优化。图 4-9 表示从 employees 数据表 emp_no 列所建的索引中只读取第一条与最后一条记录，进而获得最小值与最大值。

第二个查询中，salaries 数据表的索引由 emp_no+from_date 组成，检索 emp_no=10001 的记录，在检索结果中按升序或降序最后只访问到一个记录，所以可以使用上述优化。

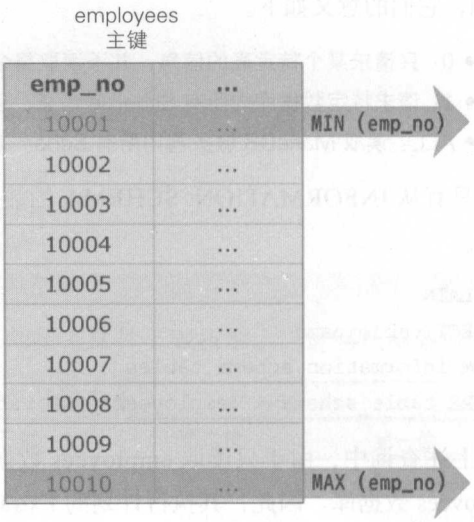


图 4-9 无 WHERE 条件的 MIN() 和 MAX() 查询优化

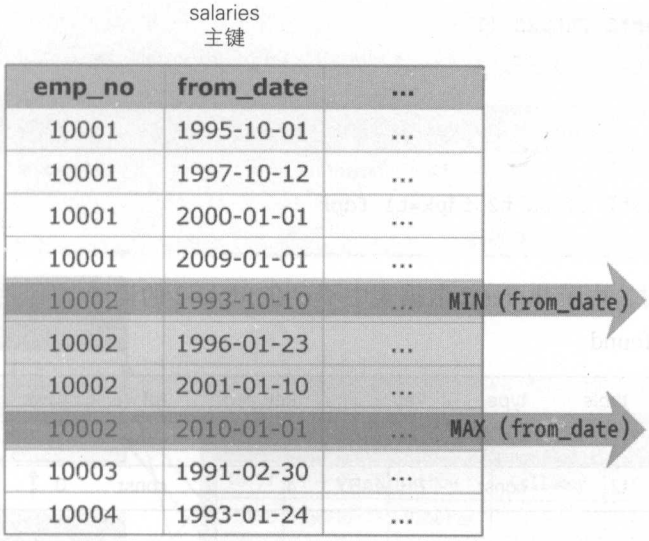


图 4-10 有 WHERE 条件的 MIN() 和 MAX() 查询优化

14. Skip_open_table、Open_frm_only、Open_trigger_only、Open_full_table

与 Scanned N databases 类似，只有 SELECT 查询访问 INFORMATION_SCHEMA 数据库的元数据信息时，其执行计划的 Extra 中才会显示标题所示信息，比如仅从存储数据表元数据信息的 *.FRM 文件与存储触发器的 *.TRG 文件或者数据文件中读取必需的文件，还是读取所有文件等。Extra 列显示的信息是以下四种之一，各自含义如下：

(1) skip_open_table：不必另外读取保存数据表元信息的文件；

(2) Open_frm_only：仅打开保存数据表元信息的文件（*.FRM）并进行读取；

(3) Open_trigger_only：仅打开保存触发器信息的文件（*.TRG）并进行读取；

(4) Open_full_table：因无法优化，将数据表元信息文件（*.FRM）、数据文件（*.MYD）、索引文件（*.MYI）全部读取。

上述内容中，有关数据文件（*.FRM）或索引（*.MYI）的内容仅针对 MyISAM，不适应于使用 InnoDB 存储引擎的数据表。

15. unique row not found

借由两个表各自的唯一列（含主键）执行外部连接的查询中，若外连表中不存在一样的记录，则在 Extra 列显示 unique row not found。

```
--// 创建用于测试的数据表
CREATE TABLE tb_test1 (fdpk INT, PRIMARY KEY(fdpk));
CREATE TABLE tb_test2 (fdpk INT, PRIMARY KEY(fdpk));

--// 向创建的数据表插入记录
INSERT INTO tb_test1 VALUES (1),(2);
```

```
INSERT INTO tb_test2 VALUES (1);

EXPLAIN
SELECT t1.fdpk
FROM tb_test1 t1
  LEFT JOIN tb_test2 t2 ON t2.fdpk=t1.fdpk
WHERE t1.fdpk=2;
```

执行上述查询时，由于 tb_test2 数据表中不存在 fdpk=2 的记录，所以执行计划的 Extra 列显示 unique row not found。

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	t1	const	PRIMARY	4	const	1	Using index
1	SIMPLE	t2	const	PRIMARY	4	const	0	unique row not found

16. Using filesort

虽然处理 ORDER BY 时可以使用索引，但无合适的索引可用时，MariaDB 服务器会再次对访问的记录进行排序。只有无法使用索引处理 ORDER BY 时，执行计划的 Extra 列才显示为 Using filesort。这意味着将访问的记录复制到用于排序的内存缓冲区，然后采用快速排序算法进行排序。Using filesort 只显示在带有 ORDER BY 子句的查询的执行计划中。

```
EXPLAIN
SELECT * FROM employees ORDER BY last_name DESC;
```

由于 last_name 列没有索引，所以无法使用索引处理查询的排序操作。MariaDB 优化器先读取记录，再将其复制到排序缓冲区（Sort buffer），排序后将结果发送给客户端。

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	employees	ALL					300584	Using filesort

若查询的执行计划的 Extra 列显示为 Using filesort，则该查询会给系统带来很重的负荷。如果可能，请先对查询进行调优，或者创建索引。Using filesort 是比较重要的内容，5.2 节会进行详细介绍。

17. Using index

若处理查询时完全不必读取数据，则可只读取索引，此时查询执行计划的 Extra 列就会显示“Using index”（覆盖索引）。使用索引处理的查询中，最占负荷的部分是检索索引，读取数据文件，以获取符合条件的记录的其余列值。在最坏的情形下，通过索引检索到的每条记录都要读取一次磁盘。

如图 4-11 所示，实际数据存储在 employees 数据表中。假如处理下列查询时使用索引范围扫描访问方式，那么从磁盘读取数据要花多长时间呢？

```
SELECT first_name, birth_date
FROM employees WHERE first_name BETWEEN 'Babette' AND 'Gad';
```

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	employees	range	ix_firstname	42		...	Using where

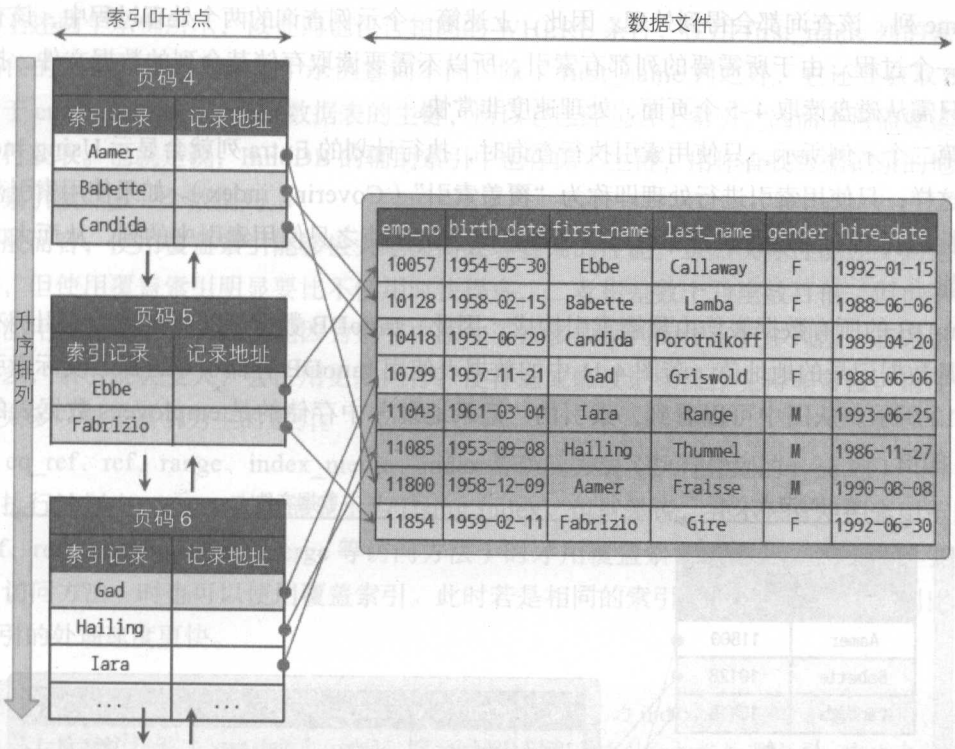


图 4-11 检索 B-Tree 索引并读取数据记录

示例查询使用 employees 数据表的 first_name 列所建的索引（ix_firstname）检索符合条件的记录。

对于符合条件的 5 条记录，为了获取相应的 birth_date 列值，需要从磁盘读取存储各条记录的数据页。

其实，为了在 ix_firstname 索引中检索到符合条件的 5 条记录，只需要读 3~4 次磁盘就能获取所需的所有索引页。但为了获取各条记录其余的数据，最多还要进行 5 次读磁盘操作。当然，示例查询很简单，只处理少量记录，所以读磁盘次数较少。但如果实际的查询很复杂，需要检索大量记录，那么读取其余记录的过程中可能还需要进行数百次读磁盘操作。

下面修改查询，删除 birth_date 列，只查询 first_name 列，同样使用索引范围扫描访问方法。

```
SELECT first_name
```

```
FROM employees WHERE first_name BETWEEN 'Babette' AND 'Gad';
```

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	employees	range	ix_firstname	42		...	Using index

employees 数据表的诸多数据列中，上述查询只用到了 first_name 列。也就是说，只要存在 first_name 列，该查询都会得到处理。因此，上述第一个示例查询的两个处理过程中，该查询只执行第一个过程。由于所需要的列都有索引，所以不需要读取存储其余列的数据文件。执行该查询时只需从磁盘读取 4~5 个页面，处理速度非常快。

如第二个示例所示，只使用索引执行查询时，执行计划的 Extra 列就会显示 Using index 信息。像这样，只使用索引进行处理即称为“覆盖索引”（Covering index）。如果使用索引范围扫描时，性能仍然不满足要求，那么可以考虑修改查询，使之只使用索引中的列，从而大大提高查询性能。

InnoDB 的所有数据表均由聚簇索引构成。因此，InnoDB 数据表的所有辅助索引拥有主键值，它是数据记录的地址值。若图 4-11 中的数据表使用 InnoDB 存储引擎，那么实际索引存储如图 4-12 所示。从图中可以看到，索引的“记录地址”中存储的是 employees 数据表的主键 emp_no 的值。

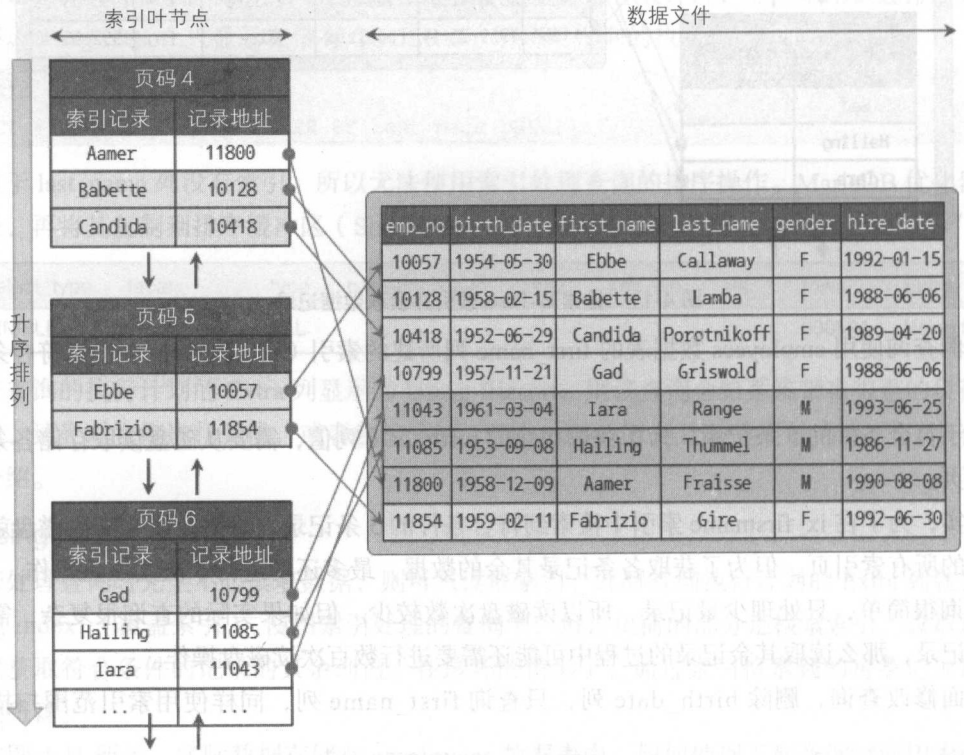


图 4-12 InnoDB 中 B-Tree 索引与数据记录

InnoDB 数据表中，即使只使用 first_name 列创建索引，最后在索引中也会同时保存 emp_no 数据列。由于这种聚簇索引特性，查询很有可能通过“覆盖索引”得到处理。下面介绍简单的查询示例，假设该示例查询也采用索引范围处理。

```
SELECT emp_no, first_name
FROM employees WHERE first_name BETWEEN 'Babette' AND 'Gad';
```

如上述两个示例所示，该查询也包含相同的 WHERE 条件，使用 first_name 列的索引检索符合条件的记录。然而与第二个示例查询不同，除了 first_name 列之外，它还要获取 emp_no 列。由于 emp_no 是 employees 数据表的主键，所以它已经包含于索引，因而不再需要读取数据文件进行获取。也就是说，InnoDB 的辅助索引中也存储了主键，用作查找数据记录的地址，这样会使索引额外多一个数据列。

一般而言，使用覆盖索引能够极大地提高处理查询的性能。虽然实际性能还受到记录条数的影响，但使用覆盖索引明显要比不使用时快得多，二者相差数十乃至数百倍。但也不能不分条件地滥用覆盖索引，否则可能因为索引中添加了过多数据列而带来更多风险。若索引中的数据列太多，索引就会变大，会占用更多内存，使存储或修改记录操作变得非常慢。请注意，不要创建太多以覆盖索引为主的索引。

像 eq_ref、ref、range、index_merge、index 等访问方法（执行计划的 type 列）一样，使用索引的执行计划中，Extra 列全部显示为 Using index。也就是说，并不只有使用索引范围扫描（eq_ref、ref、range、index_merge 等访问方法）时才用覆盖索引进行处理。执行索引全扫描（index 访问方法）时也可以使用覆盖索引。此时若是相同的索引全扫描访问方法，则比不使用覆盖索引的处理速度更快。

注意 经常有人将 Extra 列的 Using index 与访问方法（type 列值）的 index 混淆，二者概念不同，一定要注意区分。前面已经讲过，执行计划 type 列显示的 index 表示索引全扫描处理方式，它比索引范围扫描的处理速度更慢。而 Using index 信息则意味着比不使用覆盖索引的查询有更快的处理速度，覆盖索引的使用是否与执行计划的 type 列无关。

18. Using index for group-by

处理 GROUP BY 时，MariaDB 服务器需要高负荷工作。先使用分组基准列进行排序，然后将排序结果进行分组。但若使用索引（限于 B-Tree 索引）处理 GROUP BY，就会按已经排列好的顺序依次读取索引列，这样只要进行分组处理即可。像这样，使用索引处理 GROUP BY 时，由于不需要对记录排序，只读取索引必要的部分即可，所以处理速度相当快，并且非常高效。使用索引处理 GROUP BY 时，执行计划的 Extra 列就会显示 Using index for group-by 信息。处理 GROUP BY 时读取索引的方法称为“松散索引扫描”（Loose index scan），5.3.2 节将详细介绍。

处理 GROUP BY 时，简单的顺序读取索引方式与只读取索引必要部分的松散索引扫描方式是不同的。

使用紧凑索引扫描（索引扫描）处理 GROUP BY

带有 GROUP BY 子句的查询中含有 AVG()、SUM()、COUNT(*) 等函数时，处理查询时要读取所有索引。此时即便使用索引处理 GROUP BY 子句，也无法只“松散地”读取所需记录。处理这种查询时，虽然的确使用索引处理 GROUP BY 子句，但也不能称之为“松散索引扫描”，而且这种查询的执行计划中也不会显示“Using index for group-by”信息。

```
EXPLAIN
SELECT first_name, COUNT(*) AS counter FROM employees GROUP BY first_name;
```

id	select_type	table	type	key	key_len	ref	Rows	Extra
1	SIMPLE	employees	index	lx_firstname	44		299809	Using index

使用松散索引扫描处理 GROUP BY

单列索引中除了分组列外，若查询不访问任何内容，则可以使用松散索引扫描。多列索引中，GROUP BY 子句必须能使用索引。一些查询中，访问的值只要读取索引的第一个或最后一个记录即可。此时可以使用松散索引扫描，并且读取索引时只需“松散地”读取所需部分。请看下列示例查询，salaries 数据表的索引（由 emp_no+from_date 列组成）中，对于每个 emp_no 组别，通过索引读取第一个 from_date 值（最小值）与最后一个 from_date 值（最大值）即可，所以可以使用“松散索引扫描”方式处理。

```
EXPLAIN
SELECT emp_no, MIN(from_date) AS first_changed_date, MAX(from_date) AS last_changed_date
FROM salaries
GROUP BY emp_no;
```

id	select_type	table	type	key	key_len	ref	Rows	Extra
1	SIMPLE	salaries	range	PRIMARY	4		711129	Using index for group-by

要在 GROUP BY 中使用索引，首先 GROUP BY 子句中要具备使用索引的条件。还有一点也非常重要，WHERE 子句中使用的索引也对 GROUP BY 子句是否使用索引产生影响。

无 WHERE 条件子句时

若查询完全不带 WHERE 条件子句，只要 GROUP BY 子句的数据列与由 SELECT 获取的数据列具备使用“松散索引扫描”的条件即可。对于不具备上面条件的查询，可以使用紧凑索引扫描（索引扫描）或单独的排序过程进行处理。

带有 WHERE 条件子句但无法使用索引扫描时

GROUP BY 子句可以使用索引，但 WHERE 条件子句无法使用索引时，首先读取索引进行 GROUP BY，然后读取数据记录处理 WHERE 条件中的比较。因此，在此情形之下也不能使用“松散索引扫描”，而是通过紧凑索引扫描（索引扫描）过程处理 GROUP BY。请看下列查询，WHERE 子句不能使用索引，但 GROUP BY 子句可以使用索引。

```
EXPLAIN
SELECT first_name FROM employees
WHERE birth_date>'1994-01-01' GROUP BY first_name;
```

id	select_type	table	type	key	key_len	ref	Rows	Extra
1	SIMPLE	employees	index	ix_firstname	44		299809	Using where

带有 WHERE 条件子句且可使用索引搜索时

执行一个单位查询时，除 index_merge 外的访问方法只能使用一个索引。因此，WHERE 子句的条件使用索引时，GROUP BY 使用索引的条件就变得更加苛刻。也就是说，仅 WHERE 子句的条件与 GROUP BY 处理使用相同索引时，才能使用松散索引扫描。WHERE 条件句所用的索引与 GROUP BY 子句所用的索引不同时，优化器一般倾向于创建使 WHERE 条件句使用索引的执行计划。有时即使 WHERE 条件完全无法缩小范围，优化器也会指定一个执行计划，使 WHERE 条件优先 GROUP BY 使用索引。

```
EXPLAIN
SELECT emp_no
FROM salaries WHERE emp_no BETWEEN 10001 AND 200000
GROUP BY emp_no;
```

id	select_type	table	type	key	key_len	ref	Rows	Extra
1	SIMPLE	salaries	range	PRIMARY	4		207231	Using where; Using index for group-by

提示 一个查询中，即使 WHERE 子句（用索引进行检索）与 GROUP BY 子句使用相同索引，也可能不使用松散索引扫描。因为依据 WHERE 条件检索到的记录条数很少时，即使不用松散索引扫描也会有非常快的处理速度。需要对大量记录进行分组（GROUP BY）处理时，使用松散索引扫描能够得到明显的性能提升效果，优化器会根据具体情况做出适当的判断。

请看下列查询示例，它与上述查询示例几乎一样，只是进一步缩小了 WHERE 子句的检索范围，从其执行计划的 Extra 列可以看到“Using index for group-by”信息消失。

```
EXPLAIN
SELECT emp_no
FROM salaries WHERE emp_no BETWEEN 10001 AND 10099
GROUP BY emp_no;
```

id	select_type	table	type	key	key_len	ref	Rows	Extra
1	SIMPLE	salaries	range	PRIMARY	4		1404	Using where; Using index

19. Using join buffer(Block Nested Loop)、Using join buffer(Batched Key Access)

通常，为了快速执行查询，要在连接数据列上创建索引。事实上，连接所需的索引并不需要两个连接数据表中的所有列，而只需要连接操作中后读取的数据表的列。MariaDB 优化器会调查两个连接数据表中各数据列的索引，若存在无索引的数据表，则先读取后连接。这是因为后读取的数据表主要用于检索，若无索引，将会对性能产生极大影响。

进行连接时，若被驱动表的连接列有合适的索引，则不会发生任何问题。但如果被驱动表中不存在适于检索的索引，则会根据从驱动表（连接时首先要读取的数据表）中读取的记录条数，每次都对被驱动表进行全表扫描或索引全扫描。此时，为了解决被驱动表（连接中后读取的数据表）检索低效的问题，MariaDB 服务器会将从驱动表读取的记录保存到临时的内存空间，每当需要时就重新使用。用于临时保存所读取记录的内存空间称为“连接缓冲”（join buffer），使用“连接缓冲”的执行计划的 Extra 列会显示“Using join buffer”信息。

通过 join_buffer_size 系统变量可以设置用于连接的最大缓冲区大小。若连接的列有合适的索引，则可不必太担心连接缓冲区的大小。否则，最好对连接缓冲有恰当的限制，使连接缓冲在使用时不致太过不足或使用过度。用于在线 Web 服务的 MariaDB 服务器通常将连接缓冲设置为 1MB 左右即可。但从 MariaDB 5.3 开始，连接缓冲也用于执行基于块的散列连接。因此，若要执行大容量查询，像数据仓库，最好将连接缓冲区设置得更大一些。下列示例查询用于执行无条件的笛卡儿连接（Cartesian join），而执行这种笛卡儿连接的查询通常都会使用连接缓冲。

```
EXPLAIN
SELECT *
FROM dept_emp de, employees e
WHERE de.from_date>'2005-01-01' AND e.emp_no<10904;
```

上述示例查询的执行计划如下所示。

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	de	range	ix_fromdate	3		1	Using index condition
1	SIMPLE	e	range	PRIMARY	4		902	Using where; Using join buffer (Block Nested Loop)

执行计划的 Extra 字段显示“Using join buffer”信息，表示查询使用连接缓冲以执行连接操作。紧接其后的“Block Nested Loop”表示使用连接缓冲处理为嵌套循环连接（Nested loop）。Using Join buffer 是以前版本中使用的语句，而 MySQL 5.5、MariaDB 5.3 及以前版本中，由于只处理嵌套循环形态的连接，所以不需要另外显示“Block Nested Loop”语句。

但 MySQL 5.6、MariaDB 5.5 及以后版本中，除了嵌套循环连接之外，又引入了许多连接方法。因此，某些特定查询的执行计划中也会显示“Using join buffer(Batched Key Access)”字句。Batched Key Access 是从 MySQL 5.6 与 MariaDB 5.5 开始引入的新连接方法之一，详细内容将在后续章节进行讲解。此处只需知道显示在“Using join buffer”之后的内容表示使用哪种连接算法。

20. Using sort_union、Using union、Using intersect、Using sort_intersection

使用 Index_merge 访问方式（执行计划的 type 列值为 index_merge）执行查询时，可以同时使用两个以上的索引。此时为了进一步说明如何合并从两个索引读取的结果，会在执行计划的 Extra 列选择显示以下 4 种信息之一。

(1) Using intersect(...)

使用 AND 连接各个使用索引的条件时，该信息表示从各处理结果获取交集。

(2) Using union(...)

使用 OR 连接各个使用索引的条件时，该信息表示从各处理结果获取并集。

(3) Using sort_union(...)

执行的处理与 Using union 相同，但无法使用 Using union 处理时（用 OR 连接的量相对大的 range 条件），才使用该方式进行处理。Using sort_union 与 Using union 的不同在于，Using sort_union 要先读取主键，进行排序合并后，才能读取记录并返回。

(4) Using sort_intersection(...)

MySQL 5.6 中，Using intersection 执行计划只能用于相等比较条件。但从 MariaDB 5.3 开始，使用 BETWEEN 或 IN 等比较范围运算符的条件中也可以利用 sort_intersection 使用 index_merge 执行计划。sort_intersection 类似于 sort_union，也是先排序再取交集。

Using union() 与 Using sort_union() 均可用于使用 OR 连接使用索引的条件。Using union() 常用于匹配（比如相等比较 Equal）记录数不多的情形，而使用大于或小于等这类会有很多记录的条件时，常常使用 Using sort_union()。但是，无论有多少记录，若各 WHERE 条件中使用的比较条件全为相等条件，则使用 Using union()，否则使用 Using sort_union()。

提示 MariaDB 内部，Using union() 与 Using sort_union() 使用不同的排序算法。Using union() 使用单路排序算法（Single pass），而 Using sort_union() 使用双路排序算法（Two pass）。关于两种排序算法的详细介绍请参考 5.2.2 节。

21. Using temporary

MariaDB 处理查询时会使用临时表（Temporary table）存储中间结果。临时表可以在内存中创建，也可以在磁盘上创建。若查询执行计划的 Extra 列显示为“Using temporary”，则表示使用临时表。此时，仅凭执行计划本身无法判断临时表是在内存中创建的，还是在磁盘上创建的。

```
EXPLAIN
SELECT * FROM employees GROUP BY gender ORDER BY MIN(emp_no);
```

上述查询中，由于 GROUP BY 与 ORDER BY 使用不同数据列，所以执行时需要使用临时表。最具代表性的查询是无法使用索引的 GROUP BY 查询，其执行计划中会显示“Using temporary”。

Id	select_type	Table	type	key	key_len	ref	rows	Extra
1	SIMPLE	employees	ALL				300584	Using temporary; Using filesort

注意 也有很多时候，虽然执行计划的 Extra 列并未显示“Using temporary”信息，但其实其内部也使用了临时表。请注意，不要仅凭 Extra 列没有显示“Using temporary”信息而草率地断定没有使用临时表。在内存或磁盘上创建临时表的有如下这些最具代表性的查询。

- From 子句中的子查询必定创建临时表。当然有时也将其称为“派生表”（Derived table），但其实就是临时表。
- 含有 COUNT(DISTINCT column1) 的查询无法使用索引时，也会创建临时表。
- 使用 UNION 或 UNION ALL 的查询也总是使用临时表合并结果。
- 不能使用索引的排序操作也会使用临时缓冲空间，要排序的记录不断增多时，最后会使用磁盘。排序中使用的缓冲本质上就是临时表。查询执行排序操作时，其执行计划的 Extra 列显示“Using filesort”。

至于临时表或缓冲存储在内存还是磁盘，可以依据 MariaDB 服务器的状态变量的值进行判断。关于临时表的详细说明请参考 5.5 节。

22. Using where

我们已经在 MariaDB 的架构部分提到过，从内部可以将 MariaDB 大致分为 MariaDB 引擎与存储引擎两层。其中，各存储引擎负责从磁盘或内存读取所需记录，或者将记录存储到磁盘或内存；MariaDB 引擎负责加工或运算从存储引擎得到的记录。仅当 MariaDB 引擎层进行数据加工并进行过滤处理时，Extra 列才会显示“Using where”。

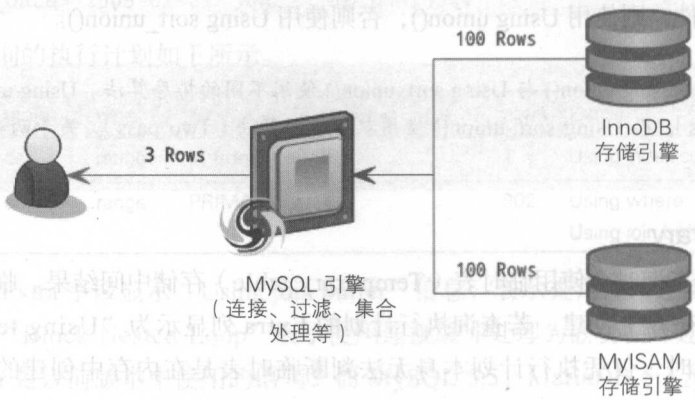


图 4-13 MariaDB 引擎与各存储引擎的处理差异

如图 4-13 所示，存储引擎总共读取了 200 条记录。MariaDB 引擎若不对它们另外进行过滤或加工，将其原封不动地传递给客户端，则不会显示“Using where”信息。其实，处理范围限制条件由各存储引擎进行处理，而检查条件由 MariaDB 引擎层处理。请看下列查询。

```
EXPLAIN
```

```
SELECT * FROM employees WHERE emp_no BETWEEN 10001 AND 10100 AND gender='F';
```

上述查询中，限制条件为 emp_no BETWEEN 10001 AND 10100，而 gender='F' 为检查条件。虽然满足第一个条件（emp_no）的记录有 100 条，但是同时满足两个条件的记录只有 37 条。存储引擎会从磁盘读取 100 条记录，然后交给 MariaDB 引擎，MariaDB 引擎会从 100 条记录中过滤掉 63 条不符合条件的记录。“Using where”即表示过滤掉不符合条件的 63 条记录。

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	employees	range	PRIMARY	4	NULL	100	Using where

MariaDB 的执行计划中，Extra 列常常显示为“Using where”，所以经常被大家忽略。其实，很多时候我们根本不理解为何要显示“Using where”。此外，MariaDB 5.0 中还存在着 Bug，即使用主键只访问一条记录，也会显示“Using where”信息。这就需要各位能够根据执行计划的 Extra 列显示的“Using where”判断它是否会引起性能问题。从 MariaDB 5.1 开始，执行计划同时会显示 Filtered 列，使得我们更容易判断是否存在性能瓶颈。关于 Filtered 列的更多内容将在 4.3.11 节进行详细介绍。

提示 我们通过上述查询示例进一步学习了索引优化相关内容。上述查询中，最终符合条件的记录虽然只有 37 条，但存储引擎仍然读取了 100 条记录，这样做相当低效。如果 employees 数据表中存在 (emp_no+gender) 索引，又会怎样呢？此时，两个条件都会被用作范围限制条件，最终只会读取所需的 37 条记录。通常，Extra 列显示“Using where”时，表示 MariaDB 引擎进行一次过滤处理，同时向存储引擎额外指派一些无意义的工作。这是 MariaDB 具有存储引擎与 MariaDB 引擎构成的二元化结构引起的。

MariaDB 引擎与存储引擎形成的二元化结构还会引起如下问题。

```
CREATE TABLE tb_likefilter (
  category INT,
  name VARCHAR(30),
  INDEX ix_category_name(category, name)
);
SELECT * FROM tb_likefilter WHERE category=10 AND name LIKE '%abc%';
```

上述示例中，category 列与 name 列被创建为索引，但 name LIKE '%abc%' 条件并不能成为处理范围的限制条件。像这些不能成为处理范围限制条件的条件，存储引擎不会使用索引进行检查，它们由 MariaDB 引擎进行处理。也就是说，存储引擎读取所有满足 category=10 的记录，然后交给 MariaDB 引擎。MariaDB 引擎使用 name LIKE'%abc%' 条件进行检查，丢弃不符合条件的记录。

假设满足 category=10 条件的记录有 100 条，其中满足 name LIKE '%abc%' 条件的记录有 10 条。那么，MariaDB 引擎为了得到 10 条符合条件的记录，要向存储引擎请求 10 倍的数据。这是一种相当不合理的处理方式，但在 MariaDB 5.0 以前版本中无法解决该问题。InnoDB 或 MyISAM 存储引擎与 MariaDB 引擎在同一进程中运行时，以上处理方式对性能的影响不是很大。但存储引擎在 MariaDB 引擎外部运行时，比如 NDB Cluster，由于还涉及网络传送问题，所以上述处理方式就对性能产生很大影响。

从 MariaDB 5.1 的 InnoDB 插件版本开始，为了解决二元化结构带来的问题，WHERE 子句中的范围限制条件与检查条件都会传递给存储引擎。存储引擎会根据这些条件只读取符合要求的记录，然后再将获得的记录交给 MariaDB 引擎，这种处理方式消除了先前的低效问题。也就是说，从 MariaDB 5.1 开始，就上述示例而言，存储引擎将只读取 10 条符合条件的记录。MariaDB 中，这种功能称为“条件下推”（Condition push down）。

23. Using where with pushed condition

执行计划的 Extra 列显示的 Using where with pushed condition 信息表示使用“条件下推”功能。从 MariaDB 5.1 开始，“条件下推”也被引入 InnoDB 与 MyISAM 存储引擎，使各存储引擎的低效问题有了相当大的改观。

然而，使用 MyISAM 或 InnoDB 存储引擎的数据表的执行计划中，不会显示 Using where with pushed condition 信息。该消息只在使用 NDB Cluster 存储引擎的数据表中才会显示。如图 4-14 所示，NDB Cluster 是一种在 MariaDB 引擎外部运行的存储引擎，存储引擎读取的记录要通过网络传递给 MariaDB 引擎。NDB Cluster 由多个节点组成，图 4-14 中，“SQL 节点”为 MariaDB 引擎，“数据节点”为存储引擎，“SQL 节点”与“数据节点”通过网络进行 TCP/IP 通信。这种架构下，若不使用“条件下推”就会出现严重的性能低下问题。

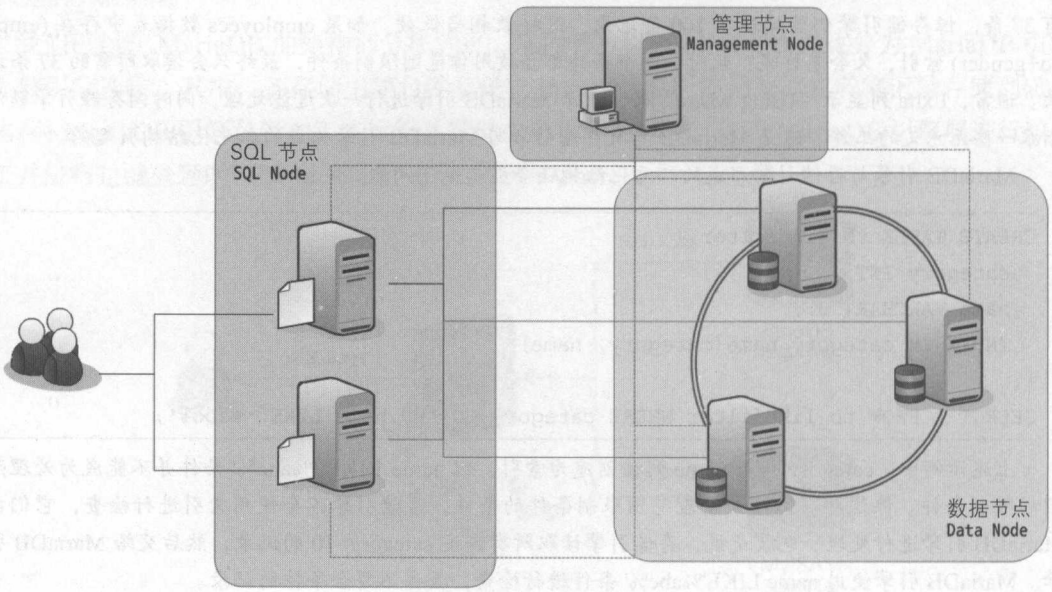


图 4-14 NDB Cluster 架构

执行计划的 Extra 列显示的 Using where with pushed condition 与前面“提示”部分讲解的“条件下推”概念十分类似，请各位参考。

24. Deleting all rows

像 InnoDB 或 MyISAM 存储引擎一样，从存储引擎的 Handler 角度看，数据表所用的存储

引擎提供删除数据表所有记录的功能时，Extra 字段就显示 Deleting all rows 信息。不带 WHERE 条件句的 DELETE 语句的执行计划中，经常出现 Deleting all rows 信息，它表示调用 1 次删除数据表所有记录的 Handler 功能（API）。以前删除数据表记录时，要依据记录条数反复调用各存储引擎的 Handler 函数，而采用 Deleting all rows 处理方式则只需调用 1 次 Handler 函数，处理起来更加快捷。

下列示例先创建名为 tab_delete 的 MyISAM 表格，再向其插入几条数据。然后查看 DELETE 语句的执行计划，DELETE 语句不带有 WHERE 条件子句，用于删除 tab_delete 数据表中的所有记录。

```
CREATE TABLE tab_delete (fd INT PRIMARY KEY) ENGINE=InnoDB;
INSERT INTO tab_delete VALUES (1), (2), (3);

EXPLAIN
DELETE FROM tab_delete;
```

上述 DELETE 语句的执行计划如下所示，可以看到在 Extra 字段显示的 Deleting all rows 信息。事实上，对于使用 Aria 存储引擎或 MEMORY 存储引擎的大部分数据表，相关查询的执行计划都会显示 Deleting all rows 信息。

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	NULL	NULL	NULL	NULL	NULL	3	Deleting all rows

但是，如果在 DELETE 语句中添加了类似 WHERE fd=1 的条件句，那么执行计划的 Extra 字段将不再显示 Deleting all rows，而显示 Using where 等对条件句进行过滤处理的信息。

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	tab_delete	range	PRIMARY	4	NULL	1	Using where

25. FirstMatch(tbl_name)

FirstMatch 是从 MariaDB 5.3 与 MySQL 5.6 开始引入的用于优化子查询的众多策略之一。下列查询的 WHERE 子句含有 IN(subquery) 类型的子查询，MySQL 5.5 或 MariaDB 5.5 以前的版本中，经常将其转换为 EXISTS 类型的查询。

```
EXPLAIN
SELECT * FROM departments d
WHERE d.dept_no IN (SELECT de.dept_no FROM dept_emp de);
```

首先查看 MySQL 5.6 中的上述查询的执行计划。

id	select_type	table	type	Key	key_len	ref	rows	Extra
1	SIMPLE	d	index	ux_deptname	122		9	
1	SIMPLE	de	ref	PRIMARY	12	d.dept_no	20696	Using index; FirstMatch(departments)

上述查询的执行计划中，Extra 字段显示 FirstMatch 关键字。FirstMatch(table) 类型的优化就是 MySQL 5.5 或 MariaDB 5.5 中使用的 IN-to-EXITST[①] 优化方式。FirstMatch(departments) 括号中的 departments 数据表是外部数据表。

MySQL 5.5 或 MariaDB 5.5 中，上述 IN(subquery) 类型的查询经常会转换为 EXISTS(subquery) 类型的查询，就像下列查询一样。其实，MySQL 5.5 或 MariaDB 5.5 中，即使用户不明确将上述查询转换为下列查询，开始执行查询之前，优化器也会先在内部进行优化转换（IN-to-EXITST 变换）。MySQL 5.6 或 MariaDB 10.0 的执行计划中，若 Extra 字段显示为 FirstMatch，则表示处理查询前要先将 IN(subquery) 查询转换为 EXISTS(subquery) 类型的查询。也就是说，先从 departments 数据表读取一条记录，然后检索 dept_emp 数据表中的记录，直到检索到符合条件的记录为止。如果在子查询中 1 次即可查找到符合条件的记录，那么执行代价会非常小；如果不存在任何符合条件的记录，那么执行代价可能会变得相当高。

```
SELECT * FROM departments d
WHERE EXISTS (SELECT 1 FROM dept_emp de WHERE de.dept_no=d.dept_no)
```

MariaDB 10.0 中的上述查询的执行计划如下所示。

id	select_type	table	Type	Key	key_len	ref	rows	Extra
1	PRIMARY	departments	Index	ux_deptname	122	NULL	9	Using index
1	PRIMARY	<subquery2>	eq_ref	distinct_key	12	func	1	
2	MATERIALIZED	dept_emp	Index	ix_fromdate	3	NULL	331603	Using index

从 MariaDB 10.0 中的查询执行计划看，与 MySQL 5.6 不同，并没有进行 FirstMatch() 类型的优化。子查询的内容首先被具体化（Materialized），然后依次连接 departments 数据表与具体化得到的临时表，再执行查询。令人遗憾的是，对于这种子查询比例较高的查询，MySQL 5.6 使用的 FirstMatch 处理方式要比 MariaDB 10.0 中使用的 MATERIALIZED 方式性能更快。但 MariaDB 10.0 并没有将这种查询引导为使用 MySQL 5.6 的 FirstMatch 处理方式的方法[②]。

MariaDB 10.0	MySQL 5.6
0.25 秒	0.00 秒

我编写了多个类型相似的查询进行测试，MariaDB 每次都采用 MATERIALIZED 方式处理，而不采用 FirstMatch 处理方式。希望发布 MariaDB 10.0 稳定版时，这些方面能够有所改善，给出更好的执行计划。

关于 FirstMatch 内容，后续子查询优化部分会再次详细讲解。

---

① MySQL 5.5 或 MariaDB 5.5 中，执行 IN(subquery) 类型的查询时，优化器在内部会将其转换为 EXISTS(subquery) 类型的查询，这种优化称为 IN-to-EXITST 优化。

② 当然，可以通过调整 MariaDB 10.0 的 optimizer_switch 诱导为 FirstMatch 处理方式，有关内容将在后文进行讲解。

26. LooseScan(m..n)

LooseScan 也是从 MariaDB 5.3 与 MySQL 5.6 开始引入的用于优化子查询的众多策略之一。IN(subquery) 类型的查询中，子查询的结果可能产生重复记录时，使用该优化方法。

本书所用的示例数据库中，要诱导使用 LooseScan 优化并非易事。下面通过略微调整 optimizer_switch 观察 LooseScan 优化。首先将 optimizer_switch 初始化为默认模式，然后关闭“firstmatch”与“materialization”优化功能，如下所示。

```
SET optimizer_switch=DEFAULT;
SET optimizer_switch='firstmatch=off';
SET optimizer_switch='materialization=off';
```

查看下列查询的执行计划。

```
EXPLAIN
SELECT * FROM departments WHERE dept_no IN (SELECT dept_no FROM dept_emp);
```

从查询的执行计划看，与 MySQL 5.5 或 MariaDB 5.2 以前版本不同，首先读取子查询中使用的 dept_emp 数据表，然后再读取 departments 数据表。此时，执行计划的 dept_emp 数据表的 Extra 字段显示 LooseScan 信息。

id	select_type	Table	type	Key	key_len	ref	rows	Extra
1	PRIMARY	dept_emp	index	ix_fromdate	3	NULL	331603	Using index; LooseScan
1	PRIMARY	Departments	eq_ref	PRIMARY	12	dept_emp.dept_no	1	

像这样，先使用 Loose Index Scan 访问方法读取子查询内容，然后删除重复记录时，要使用 LooseScan 优化方法。去除重复后的 dept_emp 数据表的结果被用作驱动表，用于与 departments 数据表连接。使用 LooseScan 处理时，不需要用另外的临时表。

关于 LooseScan 内容将在子查询优化中进行详细介绍。

注意 若不想永久修改 optimizer_switch 选项而只是暂时修改，使用后最好将其恢复为默认设置。使用下列命令可以将 optimizer_switch 恢复为默认值。

```
SET optimizer_switch=DEFAULT;
```

27. Materialize、Scan

使用具体化（Materialization）优化方法提高子查询处理速度时，执行计划就会显示 Materialize、Scan 信息。该信息只在 MySQL 5.6 与 MariaDB 10.0 的几个初始版本的 Extra 字段显示，各位使用的稳定版本中，该信息不会出现在 Extra 字段。MariaDB 10.0 与 MySQL 5.6 后期稳定版本中，使用具体化与优化时，查询执行计划的 select_type 字段将显示 MATERIALIZED 关键字。

显示为 Scan 表示不为具体化后的数据表创建任何索引，对数据表进行全扫描。若无 Scan

关键字，则表示为具体化后的临时表创建索引，并将其用作 Lookup。

28. Start materialize、End materialize、Scan

与 Extra 字段显示为 Materialize 的情形一样，Start materialize 与 End materialize 关键字只能用在 MySQL 5.6 与 MariaDB 10.0 初期版本。后期稳定版本中，在执行计划的 select_type 字段使用 MATERIALIZED 关键字进行替代。

29. Start temporary、End temporary

Duplicate Weedout 是另外一种优化子查询的方法。使用时，执行计划的 Extra 字段显示 Start temporary 与 End temporary 语句。使用 Duplicate Weedout 优化时，会先访问 IN(subquery) 查询中的子查询，然后将与外部查询数据表连接后的结果存储到临时表，之后再删除重复记录。Duplicate weedout 优化过程与连接 IN(subquery) 查询并用 GROUP BY 删除重复记录的过程一样。

下列查询只查询 dept_emp 数据表的 d001 与 d002 部门的员工。

```
EXPLAIN
SELECT * FROM employees e
WHERE e.emp_no IN (SELECT de.emp_no FROM dept_emp de WHERE dept_no IN ('d001','d003'));
```

如下所示，上述查询的执行计划中，Extra 字段显示为 Start temporary 与 End temporary。

id	select_type	table	Type	Key	key_len	ref	rows	Extra
1	SIMPLE	de	range	PRIMARY	12	NULL	74908	Using where; Using index; Start temporary
1	SIMPLE	e	eq_ref	PRIMARY	4	de.emp_no	1	End temporary

进行 Duplicate weedout 优化时，首先执行子查询，然后将结果存储到临时数据表。执行连接操作时，先读取的数据表（驱动表）的 Extra 字段显示为 Start temporary。连接中，最后读取的数据表的 Extra 字段显示为 End temporary。

30. Using index condition

执行带 WHERE 条件句的查询时，若 WHERE 子句中的条件可以使用索引，则先读取索引判断是否符合条件，需要时再读取数据文件中的记录。但 MariaDB 5.2 与 MySQL 5.5 之前版本中，即使条件可以使用索引，条件也不会被传递给存储引擎，这会导致存储引擎读取许多实际并不需要的数据文件。

请看下列查询，它从 employees 数据表中检索 first_name 列以“Lee”开始且以“matt”结尾的记录。

```
EXPLAIN
SELECT * FROM employees WHERE first_name LIKE 'Lee%' AND first_name LIKE '%matt';
```

MySQL 5.5 中，上述查询的执行计划如下所示。

id	select_type	table	type	Key	key_len	ref	rows	Extra
1	SIMPLE	employees	range	ix_firstname	44	NULL	246	Using where

从执行计划可以看到，InnoDB 存储引擎会先在 employees 数据表中检索所有 first_name 列以“Lee”开始的记录，读取符合条件的记录的所有列数据。MySQL 引擎会从结果集中通过比较查找符合 first_name LIKE'%matt' 条件的记录，然后将最终结果返回用户。InnoDB 存储引擎读取 ix_firstname 索引的同时可以一并检测 first_name LIKE'%matt' 条件，但令人遗憾的是，对于所有符合 first_name LIKE'Lee%' 条件的记录，InnoDB 存储引擎都会随机读取数据文件。到 MySQL 5.5 或 MariaDB 5.2 版本为止，由于各存储引擎无法从 MySQL 引擎获取非高效使用索引的条件，所以无法进一步优化。

上述查询在 MariaDB 10.0 中的执行计划如下所示。

id	select_type	table	type	Key	key_len	ref	rows	Extra
1	SIMPLE	employees	range	ix_firstname	44	NULL	246	Using index condition

MariaDB 10.0 的执行计划中，Extra 字段的 Using where 语句不见了，取而代之的是 Using index condition 语句。这表示将 first_name LIKE'Lee%' 与 first_name LIKE'%matt' 两个条件全都传递给 InnoDB 存储引擎。InnoDB 存储引擎使用这两个条件，最大限度地灵活使用索引，防止从 employees 数据表读取多余的数据文件。这种优化称为“索引条件下推”（Index Condition Pushdown，ICP）。

注意 请不要将 Using index condition 与 MySQL Cluster（NDB）执行计划中的 Using where with pushed condition 相混淆。虽然它们的思路类似，都是将 WHERE 条件传递给存储引擎以优化查询的执行，但 MySQL Cluster 更早引入这种优化。Using where with pushed condition 只出现在 MySQL Cluster 中，而我们常用的 MariaDB 或 MySQL 中显示 Using index condition。

31. Rowid-ordered scan、Key-ordered scan

“多范围读”（Multi Range Read，MRR）是 MySQL 5.6 与 MariaDB 5.3 开始新增的优化功能。实际应用中，有时需要先通过索引范围扫描查找符合 WHERE 条件的记录，然后再根据需要读取数据文件的其余记录。通过索引查找到的记录相当多时，对相关记录数据文件的读取每次都采用随机访问方式。事实上，这种随机访问方式相当耗费系统资源，增加系统负担，磁盘 I/O 在很多数据库中都是相当耗时的。

引入 MRR 技术正是为了解决这一问题。先通过索引读取一定量符合 WHERE 条件的记录，然后使用主键值全部排序，再从实际数据文件中读取其余数据列。通过这种方式可以有效减少随机读取数据文件的次数。

在 MySQL 5.6 中使用 Multi Range Read 时，Extra 字段会显示 Using MRR；但 MariaDB 中，Extra 字段会显示 Rowid-ordered scan 或 Key-ordered scan。其实，MySQL 5.6 只执行类似于 MariaDB 的 Rowid-ordered scan 的优化，并未实现对 Key-ordered scan 的优化。

请看下列查询，它用于查询名字在A~B的所有员工信息。该查询使用employees数据表的ix_firstname索引检索满足WHERE条件的记录，但由于要返回给用户的信息是employees数据表的全部数据列，所以必须读取数据文件。

```
EXPLAIN
SELECT * FROM employees WHERE first_name>='A' AND first_name<'B';
```

上述查询的执行计划如下所示。

id	select_type	table	type	Key	key_len	ref	rows	Extra
1	SIMPLE	employees	range	ix_firstname	44	NULL	43230	Using index condition; Rowid-ordered scan

执行计划的Extra字段显示为Rowid-ordered scan语句，表示先使用ix_firstname索引检索符合WHERE条件的记录，然后为了读取其余数据列，在读取数据文件之前，先依据employees数据表的主键emp_no列进行排序，然后再读取数据文件。

注意 先读索引再读数据文件的查询中，并非总是使用“多范围读”这一优化方法。因此，为了创建“多范围读”（MRR）执行计划，需要调整optimizer_switch设置，如下所示。

```
SET optimizer_switch='mrr=ON';
SET optimizer_switch='mrr_sort_keys=ON';
```

我运行示例时使用的MariaDB服务器版本为10.0.6，默认设置下，mrr与mrr_sort_keys优化选项均为OFF。这可能是为了与以前的版本相兼容，也是因为它还不是正式的发布版本。

Key-ordered scan执行计划不用于无连接的简单SELECT查询，而用于使用存储引擎（如使用聚集键的XtraDB、InnoDB）数据表的主键与连接的查询。下列查询中，使用dept_emp数据表的emp_no数据列与employees数据表的emp_no数据列彼此连接。请注意，emp_no列是employees数据表的主键。

```
EXPLAIN
SELECT * FROM dept_emp de, employees e
WHERE e.emp_no=de.emp_no AND de.dept_no IN ('d001','d002');
```

上述查询的执行计划中，首先读取dept_emp数据表，再执行与employees数据表的连接。此时，用emp_no列读取employees数据表之前，先对符合de.dept_no IN('d001', 'd002')条件的dept_emp数据表的emp_no列进行排序，然后读取employees数据表。由于employees数据表的emp_no列充当employees数据表的记录地址，所以首先对它排序，之后再读取employees数据表的记录。因此，执行计划的Extra字段也会一并显示BKAH join语句。

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	de	range	PRIMARY	12	NULL	75132	Using where
1	SIMPLE	e	eq_ref	PRIMARY	4	employees.de.emp_no	1	Using join buffer (flat, BKAH join); Key-ordered scan

我们通过简单的查询示例学习了有关 Rowid-ordered scan 与 Key-ordered scan（“多范围读”执行计划）的内容。后面将进一步讲解有关“多范围读”的工作方式与 BKA（Batched Key Access）相关内容。

注意 Key-ordered scan 执行计划的示例也要在修改 optimizer_switch 设置后执行。5.9.2 节将详细讲解有关 join_cache_level 系统设置的内容。

```
SET optimizer_switch='mrr=on';
SET optimizer_switch='mrr_sort_keys=on';
SET join_cache_level=8;
```

32. No matching rows after partition pruning

MariaDB 服务器检索分区表时，会从操作目录中删除不需要使用分区键列检索的分区，这称为“分区修剪”（Partition Pruning）。MariaDB 的优化器制定查询的执行计划时，若不存在符合 WHERE 条件的分区，执行计划的 Extra 字段就会显示 No matching rows after partition pruning 语句。No matching rows after partition pruning 类似于 Impossible WHERE 语句，表示无法找到符合条件的记录，遇到时请仔细检查 WHERE 条件语句。

4.3.11 EXPLAIN EXTENDED（Filtered 列）

上一小节已经详细讲解了执行计划的 Extra 列显示的 Using where 语句的含义。即使使用 MariaDB 5.1 以后的版本，存储引擎最终也不会只读取发送给用户的记录。由于连接等多种原因，各存储引擎读取的记录仍然要交给 MariaDB 引擎进行过滤处理，这一过程中必然会发生丢弃记录的情况。但 MariaDB 5.1.12 之前版本中，经过 MariaDB 引擎过滤后，没有办法知道丢弃了多少记录，也无法知道剩下多少。

从 MariaDB 5.1.12 开始，执行计划中新增了 Filtered 列，用于告知用户过滤的执行效率。若想在执行计划中同时看到 Filtered 列，只要在 EXPLAIN 命令后面添加 EXTENDED 关键字即可。下面介绍使用 EXTENDED 关键字的执行计划示例。

```
EXPLAIN EXTENDED
SELECT * FROM employees
WHERE emp_no BETWEEN 10001 AND 10100 AND gender='F';
```

使用 EXPLAIN EXTENDED 命令查看查询的执行计划时，执行计划的 rows 列后就会出现 filtered 列，如下所示。

Id	select_type	Table	type	Key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employees	range	PRIMARY	4	NULL	100	20	Using where

执行计划的 filtered 列值是一个百分数（Percentage），表示经过 MariaDB 引擎过滤后最终剩余的记录占总记录数（存储引擎读取的记录数）的比例。上述查询的执行计划中，rows 列值为

100行，filtered列值为20%，表示存储引擎总共读取了100条记录。将它们传给MariaDB引擎，经过MariaDB引擎过滤后，仅剩20%。也就是说，MariaDB引擎过滤后，剩余的记录为20条（100条*20%）。请注意，filtered列中的数值并非真实值，它是一个由统计信息推测出的值（示例中的filtered值是为了说明方便而编造出来的）。图4-15描述了执行查询时存储引擎读取了多少记录，以及MariaDB引擎过滤掉多少记录。

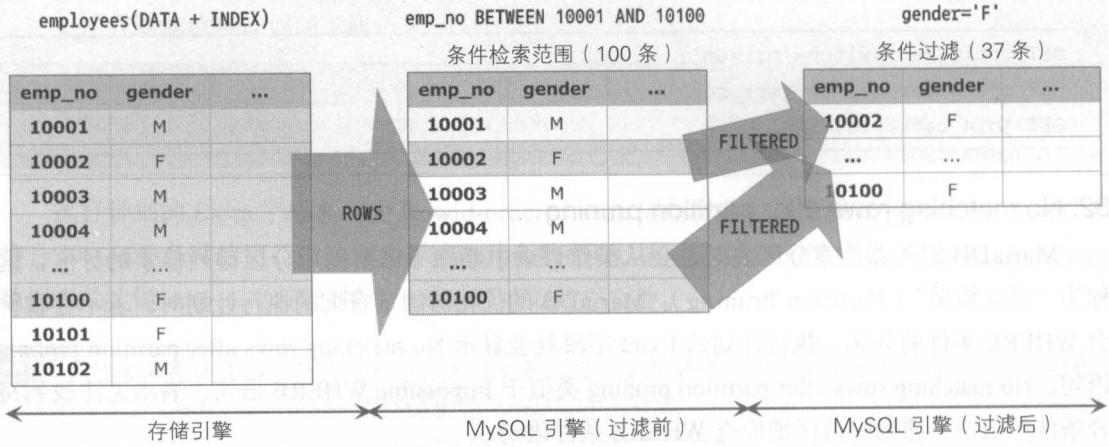


图4-15 rows与filtered列的含义

4.3.12 EXPLAIN EXTENDED（附加优化器信息）

EXPLAIN命令后使用EXTENDED选项还有一项功能。MariaDB引擎制定查询的执行计划时，会分析查询语句并创建分析树（parse tree）。一部分优化处理是使用分析树进行的。EXPLAIN EXTENDED命令的另一项功能是重组分析树，然后以查询语句的顺序显示。请看下列示例。

```
EXPLAIN EXTENDED
SELECT e.first_name,
  (SELECT COUNT(*) FROM dept_emp de, dept_manager dm WHERE dm.dept_no=de.
dept_no) AS cnt
FROM employees e
WHERE e.emp_no=10001;
```

执行EXPLAIN EXTENDED命令后，也只在画面中显示查询的执行计划，就像不使用EXTENDED选项一样。使用EXPLAIN EXTENDED命令显示执行计划后，再使用SHOW WARNINGS命令可以查看到优化器分析重组后的查询语句，如下所示。

```
MariaDB> SHOW WARNINGS;
SELECT 'Georgi' AS 'first_name',
 (SELECT COUNT(0)
  FROM 'employees'.'dept_emp' 'de'
```

```
    JOIN 'employees'.'dept_manager' 'dm'
  WHERE ('employees'.'de'.'dept_no' = 'employees'.'dm'.'dept_no')) AS 'cnt'
FROM 'employees'.'employees' 'e' WHERE 1
```

使用 SHOW WARNINGS 命令显示的内容不是标准的 SQL 语句。输出结果与示例相当类似，但由于其中包含标记形式的优化信息，所以很多时候很难把握其所表示的含义。上述示例中的 COUNT(*) 在内部被处理为 COUNT(0)，优化器会事先执行 emp_no=10001 条件，所以使用了 Georgi 常量值。

使用 EXPLAIN EXTENDED 命令能够帮助我们掌握优化器如何分析并转换查询，以及执行哪些特殊处理等，请各位记住该命令。

4.3.13 EXPLAIN PARTITIONS（Partitions 列）

PARTITIONS 是 EXPLAIN 命令的另一个选项，使用该选项可以更详细地查看分区表的执行计划信息。只使用 EXPLAIN 命令将无法查看分区表是如何使用的，但如果在 EXPLAIN 命令后使用 PARTITIONS 选项，则可以查看执行查询时使用了哪些数据表分区等信息。请看下列示例。

```
CREATE TABLE tb_partition (
  reg_date DATE DEFAULT NULL,
  id INT DEFAULT NULL,
  name VARCHAR(50) DEFAULT NULL
) ENGINE=INNODB
partition BY range (YEAR(reg_date)) (
  partition p0 VALUES less than (2008) ENGINE = INNODB,
  partition p1 VALUES less than (2009) ENGINE = INNODB,
  partition p2 VALUES less than (2010) ENGINE = INNODB,
  partition p3 VALUES less than (2011) ENGINE = INNODB
);

EXPLAIN PARTITONS
SELECT * FROM tb_partition
WHERE reg_date BETWEEN '2010-01-01' AND '2010-12-30';
```

上述示例的 tb_partition 数据表根据 reg_date 数据列的值按年份划分为 4 个分区，然后从数据表中查询 reg_date 列值在 2010-01-01~2010-12-30 的记录，使用 EXPLAIN PARTITIONS 命令查看其执行计划。查询要访问的是 2010 年全年的数据，从表创建中可知，这些数据位于表的第三个分区 p3。其实，优化器处理该查询时只要读取 p3 分区，制定访问 p3 分区的执行计划即可。像这样，从多个分区组成的数据表中删除不需要的分区，只选择查询要访问的分区，该过程称为“分区修剪”。

只有通过查询的执行计划明确要读取哪个分区，才能对查询进行调优。使用 EXPLAIN

PARTITIONS 命令可以查看优化器为执行查询而访问的数据表。使用 EXPLAIN PARTITIONS 命令显示的执行计划含有名为 partitions 的新列，在 partitions 列显示查询使用的分区目录。从下列执行计划中可以看到，上述示例查询只访问 p3 分区，和预想的一样。

id	select_type	table	partitions	type	key	key_len	ref	rows	Extra
1	SIMPLE	tb_partition	p3	ALL				2	Using where

EXPLAIN PARTITIONS 命令提供的信息可以帮助我们判断运行于分区表的查询对分区功能灵活运用的程度。EXPLAIN 命令中，无法同时使用 EXTENDED 与 PARTITIONS 选项。

注意 输入日期格式错误时，TO_DAYS() 函数会返回 NULL 值。MariaDB 的分区键使用类似 TO_DAYS() 这样可能返回 NULL 值的函数时，查询执行计划的 partitions 列也可能包含数据表的第一个分区。使用范围分区（Range Partition）的数据表中，NULL 值总是被存储到第一个分区，所以执行计划的 partitions 列也会包含第一个分区。即使 partitions 列同时包含实际所需分区与表的第一个分区，也不会影响查询性能，请各位不必太担心。

4.4 优化器提示 MariaDB MySQL

随着 MariaDB 版本升级，其统计信息与优化器的优化方法变得更加多样，查询执行计划的优化也更加成熟。即便如此，与服务开发人员或 DBA 相比，MariaDB 服务器无法保证 100% 地理解用户请求的服务，所以有时创建的执行计划并不尽如人意。此时，应该有一种方法告知优化器如何制定查询的执行计划。为此，RDBMS 提供了优化器提示功能。虽然种类不多，MariaDB 也提供了几种必需的优化器提示。

4.4.1 提示的使用方法

不同类型的 MariaDB 的优化器提示所使用的位置是固定的，下面通过示例逐一讲解。提示的标记方法大致分为如下两种，若使用不正确，则会引发错误。因为 MariaDB 不像 Oracle 一样将提示解析为注释的一部分，而是解析为 SQL 的一部分（即使提示标记于注释内）。

```
SELECT * FROM employees USE INDEX (primary) WHERE emp_no=10001;
SELECT * FROM employees /*! USE INDEX (PRIMARY) */  WHERE emp_no=10001;
```

第一个示例并未使用注释标记，而直接将提示作为 SQL 语句的一部分；第二个示例中，MariaDB 将提示以注释形式内嵌于 SQL 语句。第二种注释形式的提示中，开始标记（/*）与“!”之间没有空格，最后以 */ 标记结束。其他 DBMS 中，执行查询时，提示会被处理为注释；而 MariaDB 中，提示会被解析为 SQL 语句的一部分（注释中的提示语句），一旦使用不正确就会引发错误并终止查询。

下列标记法与优化器提示虽然关系不大，但还是希望各位牢记。

```
CREATE /*!32302 TEMPORARY */  TABLE temp_emp_stat
(hire_year INT NOT NULL, emp_count INT, PRIMARY KEY (hire_year) );

CREATE TEMPORARY TABLE temp_emp_stat
(hire_year INT NOT NULL, emp_count INT, PRIMARY KEY (hire_year) );
```

上述语句编写时采用了与优化器提示一样的标记法。其实，上述语句使用的 TEMPORARY 关键字不是提示，而是 SQL 语句的一部分，注释起始标记（/*）后的数字表示 MariaDB 的版本。“32302”代表 MariaDB 版本号为 3.23.02[①]，表示执行查询的 MySQL 版本号在 3.23.02 以上时，上述两个示例查询会执行相同功能。而低于 3.23.02 的版本中，第一个示例查询会将 temp_emp_stat 表创建为普通数据表，而非临时表（Temporary），第二个查询会尝试创建临时数据表（Temporary）。由于从 MySQL 3.23.02 开始添加了临时数据表功能，所以执行第二个查询时会发生错误。上述示例中的注释是特定于指定版本的（Version specific comment）。也就是说，向 SQL 语句应用特定功能时，该功能仅在指定版本以上的版本中才能使用。此时就要如示例所示，在注释中明确指出需要使用的 MariaDB 最低版本号。

MariaDB 版本标记法与 Unix 软件类似，大致分为三个部分：第一个句点前面的数字是主版本号（Major version），第一个句点与第二个句点之间的数字是次版本号（Minor version），第二个句点之后的数字是修订版本号（Patch version）。提示中，版本号数字的产生方法是从软件的主版本号中取一位数字，从次版本号与修订版本号中分别取两位数字。比如所用 MariaDB 版本号为 5.5.8，则提示应写为“/*!50508 TEMPORARY */”。到目前为止，MariaDB 最大版本号为 5.5.x，所以提示中关于版本号数字的最大值为“/*!50599 … */”。但即使提示中使用了更大的数字，也不会发生错误。像这样指定版本的注释还有其他用途，一般来说，类似于 /*…*/ 的注释在开始处理前，MariaDB 服务器会将其全部删除；而类似 /*!99999…*/ 的注释，MariaDB 服务器不会删除。该功能有时会为 DBA 或开发人员带来极大便利。

4.4.2 STRAIGHT_JOIN

对 MariaDB 查询调优时，与使用索引的提示相比，STRAIGHT_JOIN 提示更常用。STRAIGHT_JOIN 是优化器提示，同时也是连接关键字（如 JOIN UPDATE 或 JOIN DELETE 中看到的一样）。SELECT、UPDATE、DELETE 查询要连接多个数据表时，STRAIGHT_JOIN 可以固定连接顺序。下列查询连接了 3 个数据表，但无法判断哪个数据表是驱动表，哪个数据表是被驱动表。优化器会根据每时每刻各数据表的统计信息与查询条件按最优顺序连接数据表。

```
SELECT *
FROM employees e, dept_emp de, departments d
WHERE e.emp_no=de.emp_no AND d.dept_no=de.dept_no;
```

① 由于 MariaDB 是从 5.1 版本开始发布的，所以当然不会有 MariaDB 3.23.2 版本。但是与 MySQL 类似，MariaDB 仍能够判断 3.23.2 版本比当前 MariaDB 服务器版本号低还是高。

上述查询的执行计划如下所示，从中可知，departments 数据表被选为驱动表。首先读取它，然后读取 dept_emp 数据表，最后读取 employees 数据表。一般而言，用于连接的数据列索引是否可用将决定连接顺序。连接数据列的索引无任何问题时，选择（有 WHERE 条件时满足 WHERE 条件的）记录数少的数据表为驱动表。上述示例查询中，departments 数据表的记录最少，因而被选为驱动表。

id	select_type	table	type	..	Key	key_len	ref	rows	Extra
1	SIMPLE	d	index		dept_name	122		9	Using index
1	SIMPLE	de	ref		PRIMARY	12	employees.d.dept_no	18436	
1	SIMPLE	e	eq_ref		PRIMARY	4	employees.de.emp_no	1	

若想更改上述查询的连接顺序，可以使用 STRAIGHT_JOIN 提示。下列两个查询相同，只是提示的标记法略有不同。

```
SELECT STRAIGHT_JOIN e.first_name, e.last_name, d.dept_name
FROM employees e, dept_emp de, departments d
WHERE e.emp_no=de.emp_no AND d.dept_no=de.dept_no;

SELECT /*! STRAIGHT_JOIN */  e.first_name, e.last_name, d.dept_name
FROM employees e, dept_emp de, departments d
WHERE e.emp_no=de.emp_no AND d.dept_no=de.dept_no;
```

STRAIGHT_JOIN 提示会引导优化器按照 FROM 子句中数据表的顺序执行连接。从该查询的执行计划看，的确是根据 FROM 子句中数据表的顺序（employees → dept_emp → departments）进行的连接。

id	select_type	table	type	..	key	key_len	ref	rows	Extra
1	SIMPLE	e	ALL					300439	
1	SIMPLE	de	ref		ix_empno_fromdate	4	employees.e.emp_no	1	
1	SIMPLE	d	eq_ref		PRIMARY	12	employees.de.dept_no	1	

此处需要注意的是，MariaDB 中的提示比其他 DBMS 中的提示对优化器的影响更大。夸张点说，只要提示存在且不会导致查询无法执行，优化器就会盲从，根据其执行查询。请看下列查询。

```
EXPLAIN
SELECT /*! STRAIGHT_JOIN */
  e.first_name, e.last_name, d.dept_name
FROM employees e, departments d, dept_emp de
WHERE e.emp_no=de.emp_no AND d.dept_no=de.dept_no;
```

该查询选择 employees 数据表为驱动表，先与 department 数据表进行连接，再与 dept_emp

数据表连接。即使 employees 与 department 数据表直接进行连接的条件不成立，也会试图依据给定的提示执行查询，如下执行计划所示。

id	select_type	table	type	..	key	key_len	ref	Rows	Extra
1	SIMPLE	e	ALL					300439	
1	SIMPLE	d	index		dept_name	122		9	Using index
1	SIMPLE	de	eq_ref		PRIMARY	16	employees.d.dept_no, employees.e.emp_no	1	

上述示例查询中，按照 employees → dept_emp → departments 顺序连接与按照 employees → departments → dept_emp 顺序连接这两种情况下，处理记录方面有多少差别呢？由于示例只是简单地连接 3 个数据表，所以只要将执行计划的 rows 列相乘即可。请注意，rows 列中显示的数值是依据统计信息推测出来的。按照第一种顺序连接时，要处理 300439（300439*1*1）次记录；而依据第二种顺序连接时，要处理 2703951（300439*9*1）次记录。提示使用不当时，处理速度将会变得非常慢。从查询的实际执行结果看，前者的查询速度比后者大约快 5~8 倍。

只有在下列情形中无法确定连接顺序时，才使用 STRAIGHT_JOIN 提示强制指定连接顺序。

临时表（内嵌视图（inline view）或派生表）与普通表连接

此情形下，一般选择临时表作为驱动表。普通表的连接列无索引时，要选择记录数少的数据表为驱动表，并首先读取。大部分情形下，MariaDB 优化器都能确定合适的连接顺序，若想强制优化器制定不同的执行计划，只要使用提示即可。

临时表间的连接

临时表（子查询派生的表格）几乎总不带有索引，所以无论选哪个表作为驱动表先行读取都无妨，此时建议选择较小的表。

普通表间的连接

两个表的连接列都有索引或者都无索引时，选择记录数较少的数据表为驱动表更好。除此之外，建议选择连接列无索引的数据表为驱动表。

请注意，此处所说的记录数也包含可以使用索引的 WHERE 条件。换言之，满足该 WHERE 条件的记录数并非一定是数据表的所有记录条数。下列示例中，虽然 employees 数据表包含的记录更多，但由于满足 WHERE 条件的 employees 数据表的记录数少，所以选择 employees 数据表为驱动表。

```
SELECT /*! STRAIGHT_JOIN */
  e.first_name, e.last_name, d.dept_name
FROM employees e, departments d, dept_emp de
WHERE e.emp_no=de.emp_no AND d.dept_no=de.dept_no AND e.emp_no=10001;
```

使用 InnoDB 存储引擎的数据表中，连接前选用索引时，尽量选择主键（主键为聚簇键）而不要选用普通的辅助索引，这样能更快地得到执行结果。

由于各人开发项目的业务不同，所以前面罗列的内容不可能100%地适用于所有情形。很多图书或相关文章都着重强调“测试”这两个字，各位阅读本书内容时，若对某些内容产生怀疑，可以亲自编写查询并进行比较测试，大家要在学习中养成这种好习惯。

4.4.3 USE INDEX/FORCE INDEX/IGNORE INDEX

更改连接顺序时，另一个经常使用的方法是索引提示。与STRAIGHT_JOIN提示不同，使用索引提示时，应该在带有欲用索引的数据表之后将提示明确表示出来。MariaDB优化器基本都能选择合适的索引使用，但存在多个类似索引（含有3~4个以上的数据列）时，MariaDB优化器有时就会选择不恰当的索引。此时，可以使用索引提示强制优化器使用指定索引。索引提示大致分为如下三种，使用这三种索引提示时，都需要将要使用的索引名称放在关键字之后，并用括号括起来。若括号内无索引名或使用了不存在的索引名，都会被视为查询语法错误进行处理。此外，对于用户未命名的主键，统一使用“PRIMARY”这一名称。

USE INDEX

USE INDEX是最常用的索引提示，它只建议MariaDB优化器使用特定表的索引。用户给出索引提示后，优化器通常会采纳，但并不总使用。

FORCE INDEX

FORCE INDEX与USE INDEX没什么不同，只是它对优化器的影响比USE INDEX更大。但由于USE INDEX提示已经对优化器有了足够大的影响，所以几乎没有必要使用FORCE INDEX。从现有经验看，如果使用USE INDEX提示无法使优化器应用指定索引，那么即使使用FORCE INDEX提示，也同样无法使优化器使用它。MariaDB 5.1以前版本中，使用不带索引名的“FORCE INDEX()”形式可以禁止优化器使用任何索引。但从MariaDB 5.1开始，使用不带索引名的索引提示会被视为语法错误，升级更新时请注意这点。

IGNORE INDEX

IGNORE INDEX功能与FORCE INDEX、USE INDEX相反，它使优化器不能使用指定索引。从MariaDB 5.1开始，可以使用IGNORE INDEX诱导进行全表扫描。

使用上面介绍的3种索引提示时，我们也可以为其明确指定用途。用途是可选项，使用索引提示时若不明确指定其用途，（可用时）则给出的索引可以用于3种不同用途。

USE INDEX FOR JOIN

此处的JOIN关键字不仅指表间连接，也用于指从数据表中检索数据。前面讲解执行计划时已经提到，MariaDB中，从一个数据表检索数据也称为JOIN，所以此处使用了FOR JOIN这一名称。

USE INDEX FOR ORDER BY

明确指出索引仅用于ORDER BY。

USE INDEX FOR GROUP BY

明确指出索引仅用于GROUP BY。

如上所述，我们可以将索引用途分为三种。如果可以在 ORDER BY 或 GROUP BY 处理中使用索引，就能获得较好的性能保障。由于优化器通常可以为索引选择合适的用途，所以不必考虑太多。

关于索引提示的理论讲解到此为止，下面通过示例学习其使用方法。

```
SELECT * FROM employees WHERE emp_no=10001;
SELECT * FROM employees FORCE INDEX(primary) WHERE emp_no=10001;
SELECT * FROM employees USE INDEX(primary) WHERE emp_no=10001;
SELECT * FROM employees IGNORE INDEX(primary) WHERE emp_no=10001;
SELECT * FROM employees FORCE INDEX(ix_firstname) WHERE emp_no=10001;
```

上述示例中，第一个到第三个查询具有相同的执行计划，它们都使用 employees 数据表的主键处理查询。不使用索引提示时，由于存在“emp_no=10001”条件，因而优化器认为使用主键是最优的。第四个查询使用 IGNORE INDEX 提示有意禁用索引。示例也只是例子而已，虽然可以忽略这些毫无来由的提示（通过主键访问，但通过全表扫描进行处理的提示），但 MariaDB 5.5 中不会使用主键范围扫描，而显示使用全表扫描的执行计划。第五个查询中，使用 FORCE INDEX 提示以使用与 WHERE 条件毫无关联的索引，因而制定执行计划时抛弃了主键而使用全表扫描。

请注意，上述示例全部只用于说明索引提示的使用方法。存在全文检索（Fulltext）索引时，MariaDB 优化器即使可以使用其他普通辅助索引（B-Tree 索引），通常仍会选择全文检索索引。对于主键或全文检索索引这类索引，优化器在选择时会带有加权值。

若在某种情形下很难判断索引的使用方法或哪种执行计划更好，建议各位不要使用提示，以免对优化器制定执行计划产生影响。因为对于大家一眼能看出的优化，MariaDB 优化器瞬间就能处理好。最优执行计划会根据数据特点时刻变化，现在使用主键是好的执行计划，但明天可能就不是了。因此，如果可能，最好让优化器根据当时的统计信息选取执行计划。最好的优化要使数据最小化，以将查询从服务中删除或不需调优。若难以实现，则要通过数据模型的简单化使查询简洁，并不使用提示。束手无策时才选择使用提示。可能由于实际环境中前面的处理要耗费相当长的时间，往往也需要处理者具有足够的能力，所以人们才如此依赖这些提示吧。

4.4.4 SQL_CACHE/SQL_NO_CACHE

MariaDB 的 SELECT 查询获取的结果会暂时存储到查询缓存（符合多个条件时）。在 SELECT 查询中可以直接使用 SQL_CACHE/SQL_NO_CACHE 提示以决定是否要将查询结果存储到查询缓存。query_cache_type 系统变量的设置决定是否要将查询结果存储到查询缓存。根据 query_cache_type 变量的设置值与查询缓存提示的使用组合，决定如下内容。当然，仅符合不同条件（查询结果的大小等）的情形才会存储到查询缓存，但此处是排除了所有不同条件的结果。

	query_cache_type 系统变量的设置值		
	0 或 OFF	1 或 ON	2 或 DEMAND
无提示	不缓存	缓存	不缓存
SAL_CACHE	不缓存	缓存	缓存
SQL_NO_CACHE	不缓存	不缓存	不缓存

若想让查询缓存根据不同查询有选择地工作，只要 jiang query_cache_type 系统变量的值设置为 DEMAND，并在各查询上使用 SQL_CACHE 提示即可。使用 MariaDB 时，大多都是在查询缓存启用状态下（query_cache_type=ON）进行服务，所以几乎不存在要使用 SQL_CACHE 提示的情形。也就是说，SQL_NO_CACHE 提示要比 SQL_CACHE 提示更常用。使用 SQL_NO_CACHE 提示并不意味着无法从查询缓存中获取结果，它只是让查询的执行结果无法存储到查询缓存。从结果看，使用 SQL_NO_CACHE 并不意味着执行查询时没有检索查询缓存的操作。SQL_NO_CACHE 主要用于比较查询性能以及进行性能分析。比如下列查询不使用查询缓存，经过一系列内部处理后，最终消耗了多少时间呢？为了得到更准确的查询执行时间，反复执行多次查询。会有怎样的结果呢？

```
SELECT COUNT(*) FROM employees WHERE last_name='Facello';
```

从查询执行结果看，第一次执行相当耗时，但从第二次开始，执行速度很快，几乎在 0.001 秒内即可获得结果。这正是由于使用了查询缓存。第一次执行查询时，由于它不在查询缓存中，故需要经过一系列处理过程（先分析查询，然后对 employees 进行全表扫描，再过滤 WHERE 条件，获取所有符合条件的记录条数，最后将结果存储到查询缓存），这样显然相当耗时。但从第二次查询执行开始，由于缓存中已经存在同一查询结果，所以不必再像第一次执行那样经过所有处理过程。因此，从第二个查询开始，处理速度就变得非常快，各位能亲身体会到这一点。但这样执行时，我们很难判断查询性能或处理方式。为了解决该问题，先在需要分析性能的查询上使用 SQL_NO_CACHE 提示，禁止对查询结果进行缓存，然后再测试查询的性能。当然，测试查询性能时还要考虑许多其他因素，但与查询缓存相关的内容是要考虑的最基本的部分。

SQL_NO_CACHE 提示只能用于 SELECT 查询语句，且要紧跟在 SELECT 关键字之后。与其他提示一样，使用 SQL_NO_CACHE 提示时，既可以采用注释形式，也可以将其作为 SQL 语句的一部分直接使用 SQL_NO_CACHE 关键字。

```
SELECT SQL_NO_CACHE COUNT(*) FROM employees WHERE last_name='Facello';
SELECT /*! SQL_NO_CACHE */ COUNT(*) FROM employees WHERE last_name='Facello';
```

4.4.5 SQL_CALC_FOUND_ROWS

使用 MariaDB 的 LIMIT 时，满足条件的记录如果多于 LIMIT 中指定的条数，MariaDB 就会依据 LIMIT 指定的条数查找符合条件的记录，然后立即停止检索。但查询中包含 SQL_CALC_

FOUND_ROWS 提示时，MariaDB 查找到 LIMIT 指定条数的记录之后，仍会继续检索，不过最终还是会根据 LIMIT 指定的条数将记录返回给用户。执行含有 SQL_CALC_FOUND_ROWS 提示的查询时，会使用 FOUND_ROWS() 函数获取满足条件的所有记录条数。请看下列示例。

```
MariaDB> SELECT SQL_CALC_FOUND_ROWS * FROM employees LIMIT 5;
+--------+------------+------------+-----------+--------+------------+
| emp_no | birth_date | first_name | last_name | gender | hire_date  |
+--------+------------+------------+-----------+--------+------------+
|  10001 | 1953-09-02 | Georgi     | Facello   | M      | 1986-06-26 |
|  10002 | 1964-06-02 | Bezalel    | Simmel    | F      | 1985-11-21 |
...
+--------+------------+------------+-----------+--------+------------+

MariaDB> SELECT FOUND_ROWS( ) AS total_record_count;
+--------------------+
| total_record_count |
+--------------------+
|             300024 |
+--------------------+
```

SQL_CALC_FOUND_ROWS 提示功能可以考虑在 Web 分页技术中使用，也许有些网站的分页已经采用了该功能。但此处想说的不是它的优点，而是想告诉大家，哪种情形下不能使用 SQL_CALC_FOUND_ROWS 提示。首先将使用 SQL_CALC_FOUND_ROWS 提示的分页处理与使用 COUNT(*) 查询的示例进行比较。

SQL_CALC_FOUND_ROWS 的使用方法

```
SELECT SQL_CALC_FOUND_ROWS * FROM employees WHERE first_name='Georgi' LIMIT 0, 20;
SELECT FOUND_ROWS( ) AS total_record_count;
```

此情形下，好像只执行 1 次查询便可获得所需的两种信息，但由于执行 FOUND_ROWS() 函数时需要再执行 1 次查询，所以总共需要执行两次查询。上述示例查询中，处理 first_name='Georgi' 条件时要使用 employees 数据表的 ix_firstname 索引，采用范围扫描方式读取实际值，而实际符合该条件的记录总数为 253 条。LIMIT 条件使查询只返回前面 20 条记录，但由于使用了 SQL_CALC_FOUND_ROWS 提示，所以要读取所有符合条件的记录。也就是说，通过 ix_firstname 索引执行 253 次查找实际数据记录的操作，这也意味着发生了 253 次随机 I/O 操作（需要等待磁头移动到指定位置）。

分为两个查询执行

```
SELECT COUNT(*) FROM employees WHERE first_name='Georgi';
SELECT * FROM employees WHERE first_name='Georgi' LIMIT 0, 20;
```

该方式也要执行两次查询。首先看第一个查询，它用于获取所有符合条件的记录数。由于 WHERE 子

句中存在 first_name='Georgi' 条件，所以对 ix_firstname 索引进行范围扫描。又由于查询不需要实际的数据记录，只要求获取记录条数，所以实际不会发生查找数据记录的随机 I/O 操作，因为第一个查询是覆盖索引查询。接下来看第二个查询，它会实际读取数据记录。正像 SQL_CALC_FOUND_ROWS 中所做的一样，执行第二个查询时，先采用范围扫描方式访问 ix_firstname 索引，然后读取实际数据记录，即要进行随机 I/O 操作。但由于查询中存在 LIMIT 0, 20 限制，所以随机 I/O 只进行 20 次，而非 253 次。

以上通过简单的示例对两种方式进行了比较，从中能很容易地判断哪种方式更快。显然，使用 SQL_CALC_FOUND_ROWS 时的处理速度会很慢，这是因为内存或 CPU 运算处理都是电器速度，而读写磁盘要涉及机械操作，机械操作要比电器操作慢得多。查询为联合查询，即用 UNION(DISTINCT) 将多个 SELECT 查询语句连接起来时，即使使用 SQL_CALC_FOUND_ROWS 提示也无法通过 FOUND_ROWS() 函数获得准确记录数。若索引或查询调优做得非常好，第二种方式要比使用 SQL_CALC_FOUND_ROWS 方式的执行速度快得多，所以建议各位尽量不要使用 SQL_CALC_FOUND_ROWS 方式。请注意，SQL_CALC_FOUND_ROWS 提示并不是用来提升查询性能的，只是为了方便开发人员。这样一来，它的速度慢一些就有情可原了。

对于 COUNT(*) 查询或读取列值的 SELECT 查询，若调优不理想，或者无适合 WHERE 条件的索引，使用 SQL_CALC_FOUND_ROWS 处理也能获得较快的处理速度。当然，这种情形下通常会有一些对查询或索引调优的方法，使用这些方法调优后，一般都能更快地产生结果。虽然也有例外，但从一般观点看，将记录计数查询与数据访问查询分开执行，其效率要比使用 SQL_CALC_FOUND_ROWS 提示高很多。

4.4.6 其他提示

除前面讲过的提示之外，还有 SQL_BIG_RESULT、SQL_SMALL_RESULT、SQL_BUFFER_RESULT、HIGH_PRIORITY 等提示。由于这些提示并不常用，故本书不再展开讲解，有时间的读者可以自己参考用户手册进行学习。

4.5 分析执行计划时的注意事项

目前为止，我们已经学习了 MariaDB 中有关查询处理方式与执行计划的内容。仅查询执行计划这一部分就包含了相当多的内容，全部记下来是非常困难的。因此，下面对分析查询执行计划时各列中需要特别注意的项目进行总结。

4.5.1 Select_type 列中需要注意的项目

DERIVED

DERIVED 是 FROM 子句中的子查询形成的临时表，该临时表可以存储在内存中，也可以存储到磁

盘。一般来说，将临时表存储在内存中不会对性能产生很大影响；但数据很大而将临时表存储到磁盘时，就会导致性能下降。

UNCACHEABLE SUBQUERY

对于在查询的 FROM 子句之外部分使用的子查询，若可能，要引导 MariaDB 优化器最大程度地进行缓存，以便再次使用。但使用用户变量或部分函数时，就无法再使用这种缓存功能。若使用了这种执行计划，建议考察是否可以删除用户变量或用其他函数替代。

DEPENDENT SUBQUERY

对于在查询的 FROM 子句之外部分使用的子查询，其自身无法执行。从外部查询获取值执行时，就会显示 DEPENDENT SUBQUERY。由于这种子查询无法先执行，且子查询要依赖外部查询结果，所以将导致整个查询性能变慢。需要检查子查询是否有必要获取外部查询的值，若可能，请去除子查询对外部查询的依赖。

4.5.2 Type 列中需要注意的项目

ALL、index

index 表示索引全扫描，ALL 代表全表扫描。它们只是对象不同，但由于都是以全部记录为对象的处理方式，因而很难快速获取结果。这两种访问方式不适用于普通的 OLTP 环境，添加新索引或更改查询条件时，最好不要使用。

4.5.3 Key 列中需要注意的项目

查询无法使用索引时，执行计划的 Key 列将不显示任何内容。此时最好添加索引或修改 WHERE 条件，以使查询可以使用索引。

4.5.4 Rows 列中需要注意的项目

Rows 列中显示的行数比查询实际获取的记录数多时，最好检查查询是否可以正常使用索引，以及索引是否由可以充分缩小处理范围的数据列构成。若索引没有效率，最好选择带有足够识别能力的数据列重新生成索引，或修改查询条件。

判断 Rows 列值时要注意，即使查询中含有 LIMIT 限制，其也会被排除在 Rows 数据列的考虑对象之外。比如在一个含有 LIMIT 1 限制条件的 SELECT 查询中，虽然只会返回一条符合条件的记录给用户，但 Rows 列中仍然有可能显示一个比较大的数字。这不会引发性能问题，且有可能是优化后的查询。

4.5.5 Extra 列中需要注意的项目

执行计划的 Extra 列主要用于显示执行查询处理的相关内容，它包含许多关于查询调优的重要内容。请先记住主要的关键字，在执行计划中看到相应关键字时再仔细研究。

1. 要检查查询是否正确地表达条件时

- Full scan on NULL key
- Impossible HAVING（MariaDB 5.1 开始）
- Impossible WHERE（MariaDB 5.1 开始）
- Impossible WHERE noticed after reading const tables
- No matching min/max row（MariaDB5.1 开始）
- No matching row in const table（MariaDB 5.1 开始）
- Unique row not found（MariaDB 5.1 开始）

若上述这些语句出现在 Extra 列中，首先要查看编写的查询是否正确地表达了条件，还要检查是否存在 Bug。此外，还要检查开发数据库中用于测试的记录是否已经准备好。上述这些语句与查询性能关系不大，只表示“无相关记录”，遇到这些语句时要集中检查查询是否引起 Bug。当然，如果查询将业务条件正确地表达出来，那么可以忽视这些语句。

2. 查询的执行计划不佳时

- Range checked for each record(index map: N)
- Using filesort
- Using join buffer（MariaDB 5.1 开始）
- Using temporary
- Using where

上述语句出现在 Extra 列中时，首先要检查查询是否可以进一步优化。最后一个 Using where 语句在大部分查询中都会显示，很容易被忽视。执行计划的 Rows 列值比实际查询（SELECT）到的记录数多得多，一定要补足，以尽量减少 Rows 列值与实际查询到的记录数间的差异。若能够将这些语句从查询执行计划中去掉，当然最好；如若不然，只要性能允许，也可以忽略。但请不要忘记，一定要仔细检查。

3. 查询的执行计划很好时

- Distinct
- Using index
- Using index for group-by

可以将上述项目视为指标，表示查询已得到优化处理。特别是第二个 Using index，表示使用覆盖索引处理查询，它是 MariaDB 能提供的最好性能。若无论如何优化查询都不会对查询性能产生影响，也可以引导优化器仅使用索引处理查询（覆盖索引）。

第5章

优化

5.1 全表扫描 MariaDB MySQL

5.2 ORDER BY 处理（Using filesort） MariaDB MySQL

5.3 GROUP BY处理 MariaDB MySQL

5.4 DISTINCT处理 MariaDB MySQL

5.5 临时表（Using temporary） MariaDB MySQL

5.6 索引条件下推 MariaDB MySQL

5.7 多范围读 MariaDB MySQL

5.8 索引合并 MariaDB MySQL

5.9 数据表连接 MariaDB MySQL

5.10 子查询 MariaDB MySQL

前面讲解的MariaDB执行计划中，有些因素对性能影响很小，或者并不常用，但也有很多因素对性能产生非常大的影响。本章将进一步讲解对查询性能产生较大影响并与执行计划相关的单位处理。

讲解内容中，除全表扫描外，其他部分都在MariaDB引擎而非存储引擎中处理。MariaDB引擎中的大部分附加处理作业都会对性能产生很大影响，令人遗憾的是，这些处理一般都会降低查询性能。若MariaDB引擎不对从存储引擎读取的记录做任何加工处理，而是将其直接返回给用户，这样的查询往往会有最好的性能，但是我们进行的大部分查询都不是这样的。把握MariaDB引擎耗费很多时间处理查询的原因，对于查询调优相当有用。

5.1 全表扫描 MariaDB MySQL

全表扫描是指，处理查询请求时不使用索引，而是从头到尾读取数据表中的所有数据。MariaDB优化器主要在下列条件下选用全表扫描。

- 数据表中的记录很少，使用全表扫描比通过索引读取更快时（一般数据表由1个页面组成）。
- WHERE或ON子句中不具备使用索引的合适条件时。
- 即使查询可以使用索引范围扫描，优化器判断符合条件的记录太多时（以通过对索引的B-Tree采样而获取的统计信息为准）。[①]

整个数据表通常要比索引大得多，从头到尾读取数据表需要相当多的读磁盘操作。因此，大部分DBMS执行全表扫描时，1次可以读取多个数据块或页面，且可以对读取多少进行调整控制。但MariaDB中，无法通过变量对执行全表扫描时1次读取几个页面进行设置，也不存在这样的设置变量。因此，很多人经常认为MariaDB执行全表扫描时，会从磁盘逐页读取。

这在MyISAM存储引擎中是正确的，而在InnoDB或XtraDB存储引擎中却是错误的。InnoDB或XtraDB存储引擎中，连续读取特定数据表的数据页面时，后台线程就会自动开始预读（Read ahead）作业。预读是指先对将来要用到的数据区域进行预测，然后在对该区域数据的请求到来之前，将数据从磁盘读取到XtraDB的缓冲池。也就是说，执行全表扫描时，最初几个数据页面由前台线程（Foreground thread，客户线程）进行读取，然后从某个时间点开始，读取作业被交由后台线程执行。后台线程从接手读取作业开始，每次读取4~8个页面，并且数量不断增加，一次最大可以读取64个数据页，然后将其存入缓冲池。由于前台线程可以获取并使用事先存储在缓冲池中的数据，所以查询的处理速度会相当快。

MariaDB的XtraDB中，可以通过系统变量设置从何时开始预读。该系统变量为innodbread

① 即使可以使用索引范围扫描执行查询，但若要读取的记录太多，优化器也会选用全表扫描。此时将max_seeks_for_key变量设置为某个特定值（N），MariaDB优化器就会忽略索引基数（Cardinality）或选择性（Selectivity），而认为最大只读取N条记录即可。也就是说，该值设置得越小，MariaDB服务器越倾向于使用索引范围扫描，而不是全表扫描。

ahead_threshold，一般保持其默认设置即可。但将 MariaDB 用作数据仓库时，要将该变量值设置得更小一些，以使预读频率更高。

5.2 ORDER BY 处理（Using filesort） MariaDB MySQL

除了要获取 1~2 条记录的查询外，大部分 SELECT 查询都要使用排序（ORDER BY）。只要不是像数据仓库一样需要访问大量数据以进行统一处理的功能，大部分查询中可能都会包含记录排序。处理排序有两种方法，一种是使用索引，另一种是执行查询时使用 Filesort 处理。

	优点	缺点
使用索引	执行 INSERT、UPDATE、DELETE 查询时，索引已经排序，只需依次读取即可，处理速度非常快	• 处理 INSERT、UPDATE、DELETE 时，需要额外添加或删除索引，所以很慢 • 索引需要更多磁盘空间 • 索引个数越多，越需要更多 InnoDB 缓冲池或 MyISAM 的主键缓存空间
使用 Filesort	• 因不必创建索引，故无使用索引时的那些缺点 • 若要排序的记录不多，在内存中进行 Filesort 处理，速度也非常快	因要在执行查询时进行排序处理，目标记录数越多，查询响应速度越慢

像使用索引排序一样，对记录排序时，并非总要进行 Filesort 排序处理。但在下面情形下，几乎不可能使用索引进行排序。

- 排序基准太多，无法依据每个基准全部创建索引。
- 要对 GROUP BY 的结果或 DISTINCT 等处理结果排序时。
- 要对临时表的结果（如 UNION 的结果）重新排序时。
- 要随机获取结果记录时。

MariaDB 中，查看执行计划的 Extra 列是否显示 Using filesort 后，可以判断是否另外进行了排序处理（不使用索引）。下面讲解 MariaDB 如何进行排序处理。理解了 MariaDB 的排序特点，就能轻松掌握如何对查询调优以获得更快速度。

5.2.1 排序缓冲

执行排序操作时，MariaDB 会另外分配内存空间进行排序处理，该内存空间称为“排序缓冲”（Sort buffer）。只有排序需要时才开辟排序缓冲空间，其大小随着要排序的记录大小而变化，它的最大值由 sort_buffer_size 系统变量设置。

查询执行完毕后，用于排序的内存空间会立即返还给系统。MyISAM 或 XtraDB 等存储引擎中，用于排序的内存空间大小由 sort_buffer_size 系统变量设置；而对于使用 Aria 存储引擎的数据表，排序所用的内存空间大小由 aria_sort_buffer_size 系统变量确定。MariaDB 存储引擎中，默认采用 Aria 存储引擎处理内部临时表，所以也要同时考虑 Aria 存储引擎的数据表的

排序缓冲。

以上简单介绍了有关排序的基本内容，下面讲解排序中可能会出现的问题。如果要排序的记录数非常少，只用分配的排序缓冲就能完成排序，那么排序的处理速度就会非常快。但如果要排序的记录数大于分配的排序缓冲空间的大小，会发生什么呢？此时，MariaDB 会将要排序的记录划分为许多片段，并将其临时保存到磁盘。

如图 5-1 所示，排序时先在内存的排序缓冲中对记录进行排序，然后将排序结果临时存储到磁盘。之后再获取下一批记录，排序后临时保存到磁盘，如此往复。最后再根据缓冲大小合并排好序的记录，完成排序处理。上述排序处理过程中，合并处理称为“多路合并”（Multi-merge），多路合并的执行次数由 Sort-merge-passes 状态变量（参考 SHOW STATUS VARIABLES; 命令）累计。

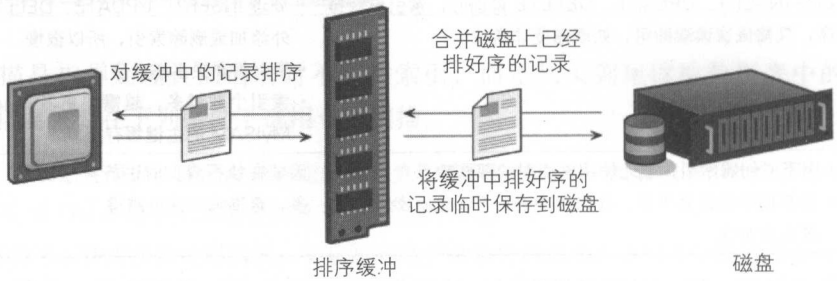


图 5-1 对排序缓冲中的记录排序并临时保存到磁盘后合并

以上这些处理都会引起磁盘读写操作，记录数越多，重复处理的次数也越多。一般认为，只要排序缓冲设置得足够大，就可以不必使用磁盘，这样能获得更快的处理速度。但实际基准测试结果表明，几乎没有差异。图 5-2 表示随着 MariaDB 的排序缓冲增加，执行查询耗费的时间。我的基准测试表明，将 MariaDB 的排序缓冲大小设置在 256 KB~512 KB 之间时，能获得最优性能，再继续增大排序缓冲也不会明显提升性能。但有些基准测试表明，将排序缓冲设置到 8 MB 以上大小时，性能又会再提升一些。此时已经不是类似 Web 的 OLTP 查询了，而是关于大容量排序处理的内容。

从现有经验看，将排序缓冲大小设置为 56 KB~1 MB 较为合适。我们可以认为，MariaDB 使用的内存分为全局内存区域与会话（本地）内存区域，用于排序的排序缓冲属于会话内存区域。也就是说，排序缓冲区域无法为多个客户所共享。同时，这也意味着连接越多，排序处理也越多，排序缓冲要耗费的内存空间就会越大。毫无依据地将排序缓冲设置得过大后——如 10~20 MB——若同时在多个连接中执行包含大量记录排序的查询，操作系统就会表现出内存不足的现象。此外，没有足够的内存空闲空间时，操作系统的 OOM-Killer 会强制终止进程，以保证有足够的内存空闲，它一般会选择占用内存最大的进程进行强制终止。一般而言，MariaDB 服务器会占用非常多的内存，所以它会是第一个被强制终止的进程。

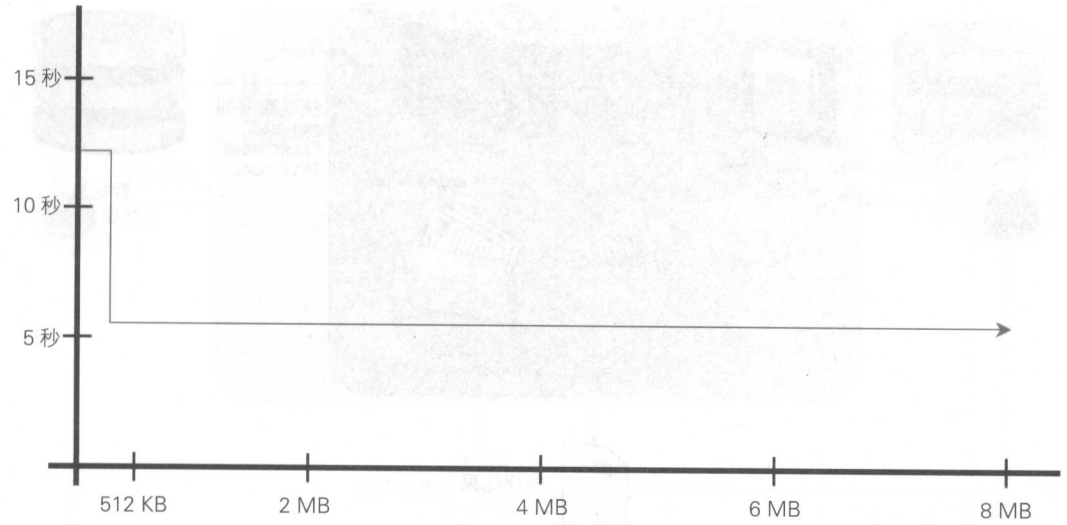


图 5-2 不同排序缓冲大小下的性能变化

注意 虽然增大排序缓冲不能明显提升性能，但可以减少磁盘读取量。因此，MariaDB 服务器的数据量很大或使用磁盘 I/O 性能低的设备时，增加排序缓冲的大小还是很有用的。但如果将排序缓冲设置得太大，导致服务器内存不足，MariaDB 服务器就会因内存不足而运行缓慢，甚至崩溃。因此，建议各位将排序缓冲设置为合适大小。

5.2.2 排序算法

根据排序时是将所有记录放入排序缓冲，还是只将排序列（用作排序基准的数据列）放入排序缓冲，我们可以将排序算法分为如下两种（非官方名称）：

一次扫描算法（Single pass）

该算法将查询（SELECT）到的所有列（含排序列）放入排序缓冲，完成排序后直接输出结果。它是 MySQL 5.0 以后引入的排序方法。

```
SELECT emp_no, first_name, last_name
FROM employees
ORDER BY first_name;
```

上述查询中，first_name 是排序基准列，emp_no、first_name、last_name 是要查询的列，使用一次扫描算法进行处理，过程如图 5-3 所示。

如图 5-3 所示，读取 employees 数据表时，将查询列（含排序列与非排序列）全部读入排序缓冲，进行排序处理。排序完成后，直接将排序缓冲中的内容返回给客户。

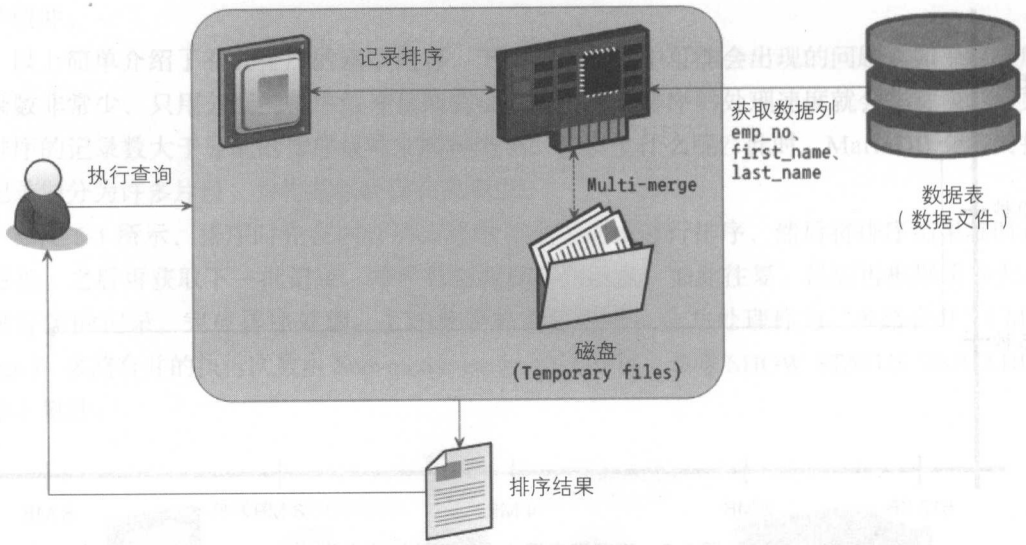


图 5-3 一次扫描排序算法

两遍扫描算法（Two pass）

排序时，只将排序列与主键值放入排序缓冲进行排序处理，然后再依据排序顺序使用主键读取数据表，最终获取要查询的数据列。该算法是 MySQL 5.0 以前版本中使用的方法。但根据要排序的记录特点，有时也会在 MariaDB 10.0 中使用。

图 5-4 表示使用两遍扫描算法（MariaDB 的旧方式）对查询进行排序的过程。首先，只从 employees 数据表读取 first_name（排序列）与 emp_no（主键）两个列，排序后，依据排序结果再次读取 employees 数据表，获取 last_name 数据列，最后将结果返回给客户。

两遍扫描算法是 MariaDB 以前使用的方式，由于要读取两次数据表（也是相同记录），所以相当不合理。而新的一次扫描排序算法不存在这种不合理的现象，但一次扫描算法需要使用更多的排序缓冲空间。若排序缓冲大小为 128 KB，使用两遍扫描算法可以对 7000 多条记录进行排序，而使用一次扫描算法大约只能对 3500 条记录进行排序。当然，根据排序缓冲空间大小与记录多少会有所不同。

最近的 MariaDB 5.x 版本中，一般使用新的一次扫描排序算法。但并非只要是 MariaDB 5.x 以上版本就总使用一次扫描排序算法。下列情形无法使用一次扫描排序算法，而要使用两遍扫描排序算法。

- 记录大小比 max_length_for_sort_data 设置值大时。
- 查询（SELECT）包含 BLOB 或 TEXT 类型的数据列时。

乍一想好像以前的排序方式速度更快，但并非总是如此。要排序的记录大小或条数较少时，使用一次扫描算法（新方式）速度会更快；而要排序的记录大小或条数相当多时，使用两遍扫描算法（旧方式）会更高效。

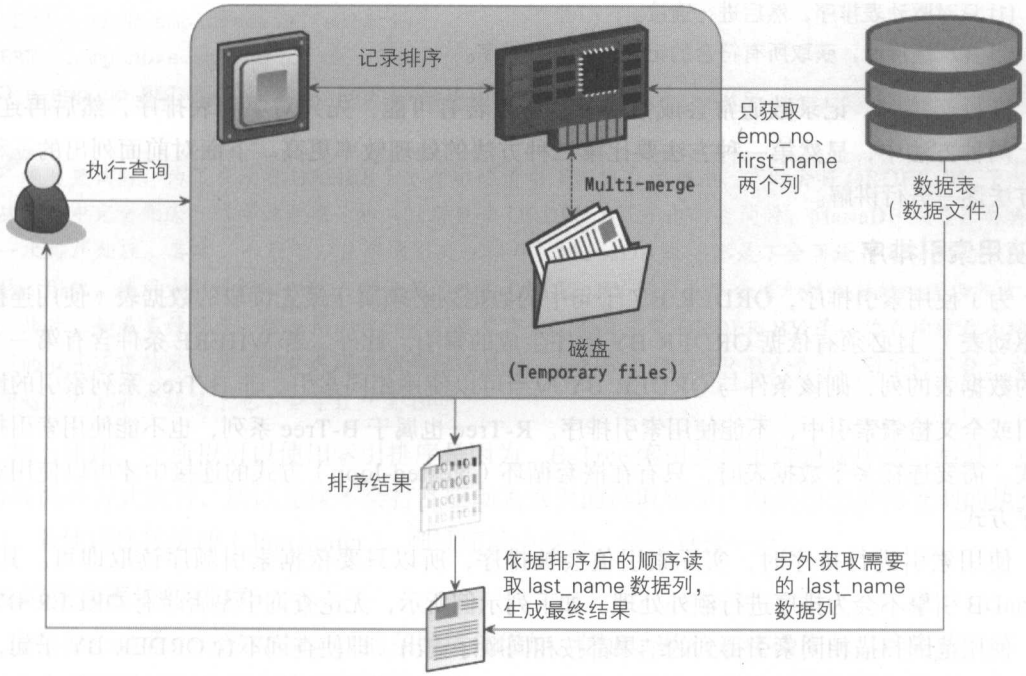


图 5-4 两遍扫描算法

注意 开发中有时需要在 SELECT 查询中获取所有列（*），而非只获取必需的列。这极有可能会降低排序缓冲的使用效率，比原来低几倍到几十倍。因此，建议各位编写 SELECT 查询时，只写必须要访问的数据列。特别是编写需要排序的 SELECT 查询时，不要写多余的数据列，这样能获得更高的执行效率。这不仅对排序缓冲有影响，对需要使用临时表的数据表也会有影响。更多内容将在 5.5 节讲解。

5.2.3 排序处理方式

处理查询中的 ORDER BY 子句时，必定使用以下三种方式之一进行排序处理。如下表所示，越往下，排序方法的处理速度会越慢。

排序处理方法	执行计划的 Extra 列
使用索引排序	无内容显示
只对驱动表排序（含无连接情形）	显示 Using filesort
将连接结果保存到临时表后，在临时表中排序	显示 Using temporary; Using filesort

首先，优化器会检查是否可用索引进行排序处理。若可用索引，则不需 Filesort 过程，依照顺序读取索引返回结果即可。但若索引不可用，则检索满足 WHERE 条件的记录，将其存储到排序缓冲，并进行排序处理（Filesort）。此时，MariaDB 优化器会采用以下两种方法之一，以尽量减少要排序的记录数。

(1) 只对驱动表排序，然后进行连接。

(2) 完成连接后，获取所有符合的记录，再进行排序。

进行连接时，记录数通常会成倍增长，所以若有可能，先只对驱动表排序，然后再连接。以上两种方法中，显然第一种方法要比第二种方法的处理效率更高。下面对前面列出的三种排序方法逐一进行讲解。

1. 使用索引排序

为了使用索引排序，ORDER BY 子句中的数据列必须属于最先读取的数据表（使用连接时指驱动表），且必须有依据 ORDER BY 顺序生成的索引。此外，若 WHERE 条件含有第一个读取的数据表的列，则该条件与 ORDER BY 应当可以使用相同索引。非 B-Tree 系列索引的散列索引或全文检索索引中，不能使用索引排序。R-Tree 也属于 B-Tree 系列，也不能使用索引排序方式。需要连接多个数据表时，只有在嵌套循环（Nested-loop）方式的连接中才可以使用索引排序方式。

使用索引进行排序时，实际索引值会被排序，所以只要依据索引顺序读取即可。其实，MariaDB 引擎不会为排序进行额外处理。如下列示例所示，无论查询中是否带有 ORDER BY 子句，使用范围扫描相同索引得到的结果都按相同顺序输出。即使查询不含 ORDER BY 子句，也要进行连接。原因在于，执行查询时要先读取 employees 数据表的主键，然后再连接 salaries 数据表，如图 5-5 所示。

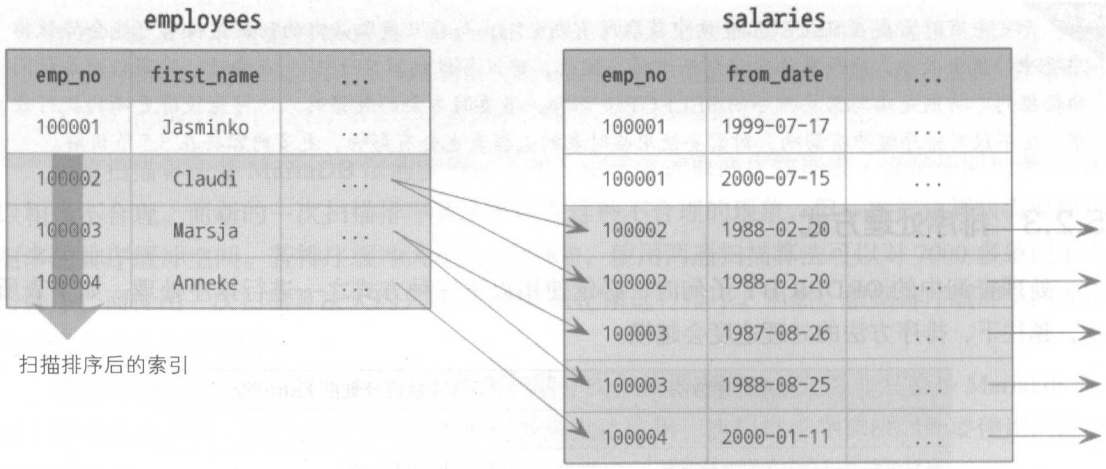


图 5-5 使用索引排序

```
SELECT * FROM employees e, salaries s
WHERE s.emp_no=e.emp_no
AND e.emp_no BETWEEN 100002 AND 100020
ORDER BY e.emp_no;
--// 使用 emp_no 列排序时，如果想使用索引自动排序而
--// 故意删除 ORDER BY emp_no 子句，这并非好的选择。
```

```
SELECT * FROM employees e, salaries s
WHERE s.emp_no=e.emp_no
AND e.emp_no BETWEEN 100002 AND 100020;
```

注意 编写查询时，为了在没有 ORDER BY 子句的情形下实现自动排序，有时会将 ORDER BY 子句本身从查询中完全删除。这样做是因为担心处理包含 ORDER BY 子句的查询时，MariaDB 服务器会再进行一次排序处理。其实，如果可以使用索引处理排序，MariaDB 服务器是不会再进行不必要的排序处理的。因此，使用索引进行排序时，即使查询包含 ORDER BY 子句也不会增加服务器的处理工作量。

此外，如果某种原因导致查询的执行计划发生变化，那么不带 ORDER BY 子句的查询就无法按照希望的顺序获得结果，这可能导致应用程序出现 Bug。如果带上 ORDER BY 子句，则不仅不会产生性能问题，而且异常状况下也不会导致产生 Bug。

如上所述，之所以可以使用索引排序是因为，B-Tree 索引是根据键值排序的。另外，由于以内嵌循环方式执行，所以连接不会打乱驱动表索引的读取顺序。但使用连接的查询的执行计划中，若使用连接缓冲（Join buffer），则顺序就会变乱，请注意这一点。

2. 仅对驱动表进行排序

连接操作通常会导致结果记录的条数成倍增加。因此，执行连接操作前，首先要对第一个数据表中的记录进行排序，然后再进行连接。这比对整个连接结果进行排序更有效率。使用这种方法的查询中，ORDER BY 子句只包含首先要读取的数据表（驱动表）的列。

```
SELECT * FROM employees e, salaries s
WHERE s.emp_no=e.emp_no
  AND e.emp_no BETWEEN 100002 AND 100010
ORDER BY e.last_name;
```

上述查询中，由于 WHERE 子句具备如下两个条件，所以优化器会选择 employees 数据表作为驱动表。

(1) WHERE 子句中的检索条件“emp_no BETWEEN 100001 AND 100010”使用 employees 数据表的主键检索可以减少工作量。

(2) 被驱动表（salaries）的连接列 emp_no 上有索引。

虽然检索时可以使用索引范围扫描，但由于 ORDER BY 子句中的数据列与 employees 数据表的主键毫无关联，所以无法使用索引进行排序。然而，已知 ORDER BY 子句中的排序基准列是驱动表（employees）中的数据列。优化器只检索驱动表，先进行排序，然后再将结果与 salaries 数据表进行连接。

图 5-6 表示该过程。

A) 使用索引检索满足 emp_no BETWEEN 100001 AND 100010 条件的 9 条记录；

B) 使用 last_name 列对检索结果进行排序（Filesort）；

C) 依次读取排好序的结果，与 salaries 数据表进行连接，最终获得 86 条记录（图 5-6 右侧的编号表

示记录连接显示的顺序）。

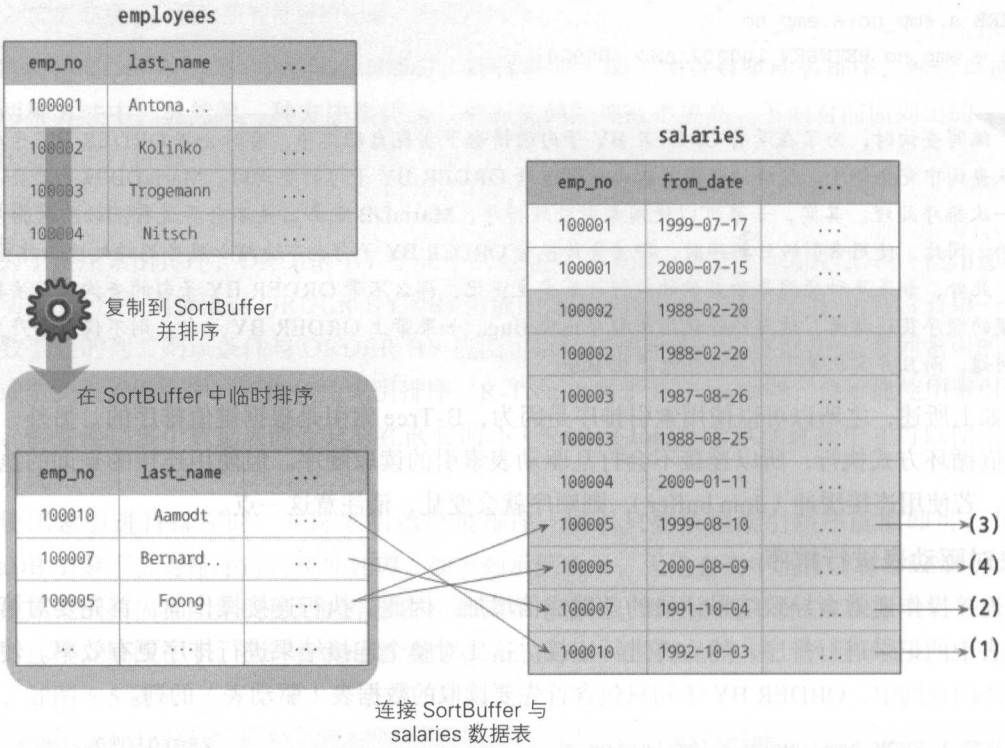


图 5-6 仅对连接的第一个数据表（驱动表）进行排序

3. 使用临时表排序

若查询不涉及多个数据表连接，只是从一个数据表中查询并进行排序，则不必使用临时表。然而，需要连接 2 个以上的数据表并对结果进行排序时，就要用到临时表。前文虽然涉及了 2 个以上数据表的连接，并进行了排序，但没有使用临时表。但除此之外的其他查询中，总是先要将连接结果存储到临时表，然后再对结果进行排序。3 种排序方法中，使用临时表排序是最慢的，因为要排序的记录数会变得非常多。下列示例查询除了 ORDER BY 子句外，其他部分与前面示例查询是完全一样的。基于相同原因，该示例查询也将 employees 数据表用作驱动表，将 salaries 数据表用作被驱动表。

```
SELECT * FROM employees e, salaries s
WHERE s.emp_no=e.emp_no AND e.emp_no BETWEEN 100002 AND 100010
ORDER BY s.salary;
```

注意观察示例查询的 ORDER BY 子句，可以发现，排序基准列不在驱动表中，而在被驱动表（salaries）中。也就是说，排序前一定要读取 salaries 数据表，所以只能对连接后的数据进行排序。

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	e	range	PRIMARY	4		9	Using where; Using temporary; Using filesort
1	SIMPLE	s	ref	PRIMARY	4	e.emp_no	4	

从查询的执行计划看，Extra 列中显示为“Using temporary; Using filesort”，表示先将连接结果存储到临时表，然后再对结果进行排序处理。图 5-7 表示整个查询的处理过程。

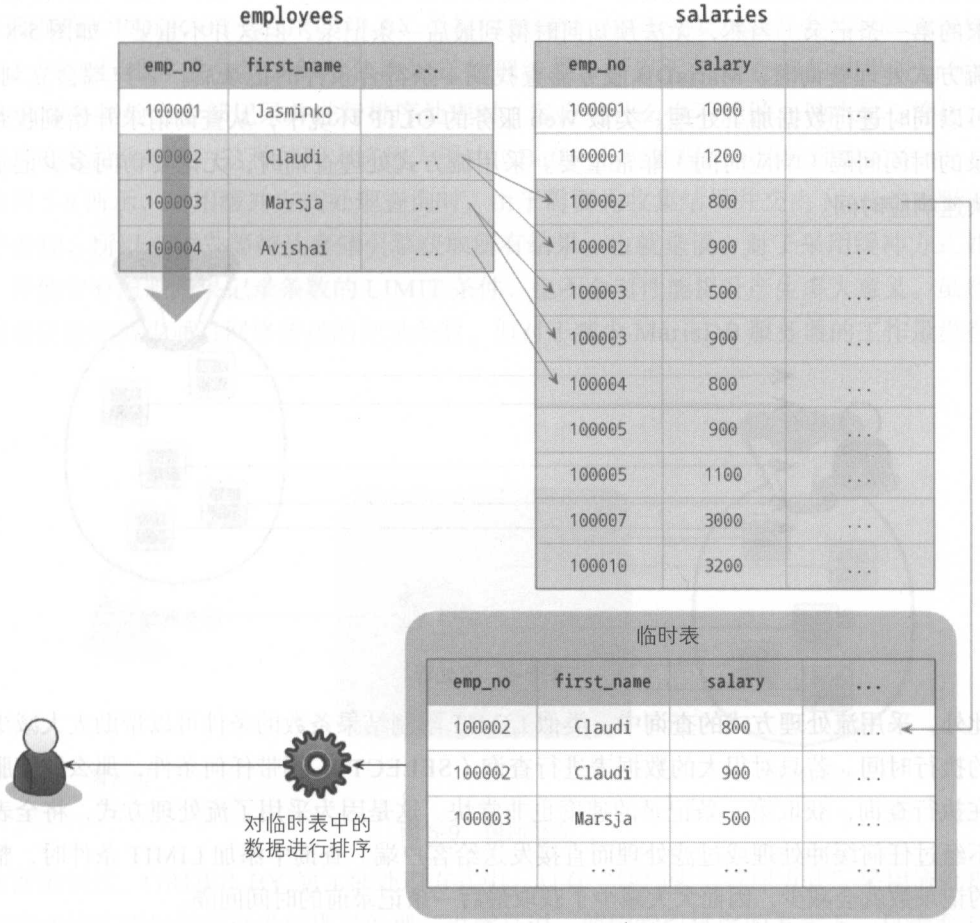


图 5-7 使用临时表排序

4. 排序方式性能比较

网页服务中使用的查询几乎都带有 ORDER BY 与 LIMIT 子句。一般而言，LIMIT 只需获取数据表或处理结果的一部分即可，因而能有效减少 MariaDB 服务器的工作量。但处理 ORDER BY 或 GROUP BY 时，不能只从满足 WHERE 条件的记录中获取 LIMIT 指定的条数。首先要获取所有满足条件的记录，进行排序或分组后，才能进行 LIMIT 记录条数限制处理。无

论 WHERE 条件中的索引优化得多好，若 ORDER BY 或 GROUP BY 使用不当，就会让查询变得缓慢。

查询中无法使用索引的排序或分组处理为何会使查询变慢呢？下面分析其中原因，为此要区分两种查询处理方法：流处理与缓冲处理。

(1) 流处理方式

如图 5-8 所示，流（Streaming）处理方式下，无论服务器端有多少数据要处理，每检索到 1 条满足条件的记录就立即发送给客户端。使用该方式处理查询时，客户端会马上得到符合查询要求的第一条记录。当然，无法预知何时得到最后一条记录，但这并不重要。如图 5-8 所示，使用流方式处理查询时，MariaDB 服务器查找到 1 条符合条件的记录后，客户端会立刻收到，因而可以同时进行数据加工处理。类似 Web 服务的 OLTP 环境中，从查询请求开始到收到第一条记录的时间间隔（响应时间）非常重要。采用流方式处理查询时，无论要访问多少记录都能保障快速响应时间。

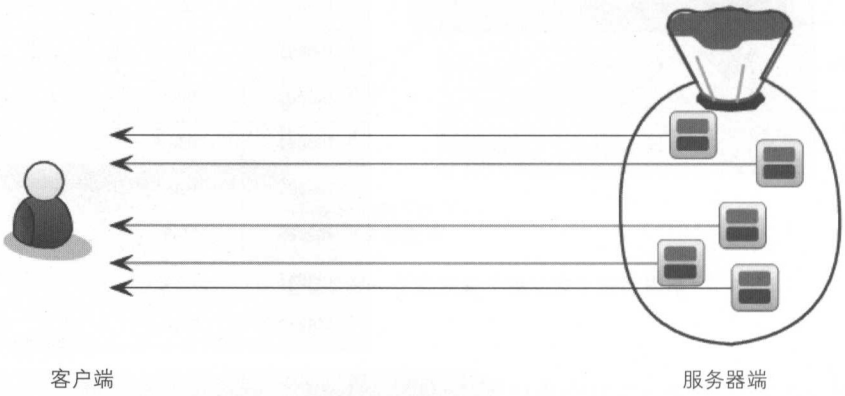


图 5-8 流处理

此外，采用流处理方式的查询中，类似 LIMIT 限制结果条数的条件可以帮助大大减少整个查询的执行时间。若只对很大的数据表进行查询（SELECT），不带任何条件，那么即使服务器仍然在执行查询，获取第一条记录的速度也非常快。这是因为采用了流处理方式，将全表扫描结果不经过任何缓冲处理或过滤处理而直接发送给客户端。查询中添加 LIMIT 条件时，整个要获取的记录数就会减少，因而大大减少了获取最后一条记录前的时间间隔。

注意 根据所用客户端工具与 API 的不同，流处理方式可能有所不同。典型的例子是使用 JDBC 库执行类似“SELECT * FROM tb_bigtable”的查询。MariaDB 服务器一旦开始读取记录，便会立即向客户端发送结果。但 JDBC 首先会将从 MariaDB 服务器接收的所有记录存储到自身缓冲，一直等待收到最后一条记录后，才将接收的所有结果返回给客户端应用程序。也就是说，虽然 MariaDB 服务器采用流方式处理并立即返回查询结果，但客户端的 JDBC 库会对返回的结果进行缓冲处理。然而，对于不使用 JDBC 的 SQL 客户端工具，由于不具有这种缓冲功能，所以无论访问的数据表有多大，都能快速获取第一条记录。

JDBC 库本身要对记录进行缓冲，原因在于，这样可以提高整个系统的吞吐量（Throughput）。这种方式下，JDBC 库与 MariaDB 服务器并非采用对话方式交换数据。无论数据大小，MariaDB 服务器总会发送数据，JDBC 从 MariaDB 服务器接收数据并保存起来。由于将不必要的网络请求减至最低，所以具有非常高的吞吐量。

JDBC 的缓冲处理方式是默认的，需要获取大量数据时，MariaDB 服务器与 JDBC 间的传送方式将变为流处理方式。

(2) 缓冲处理方式

处理带有 ORDER BY 或 GROUP BY 子句的查询时，由于先要获取符合 WHERE 条件的所有记录，再进行排序或分组，然后依次发送处理结果，所以无法对查询的结果进行流处理。MariaDB 服务器检索全部记录并进行排序处理时，客户端什么也不能做，只能耐心等待，这会使响应速度变得比较慢。这种处理方式称为缓冲（Buffering）处理，与流处理方式相对。

如图 5-9 所示，采用缓冲方式处理查询时，由于需要先收集结果并交由 MariaDB 服务器进行集中处理，所以必须先等待从存储引擎获取所有结果。也就是说，对于采用缓冲方式处理的查询，即使含有限制结果记录条数的 LIMIT 条件，也不会对性能提升产生多大意义。虽然这样的限制条件能够减少通过网络传送的记录条数，但对于减少 MariaDB 服务器的工作量没有明显作用。

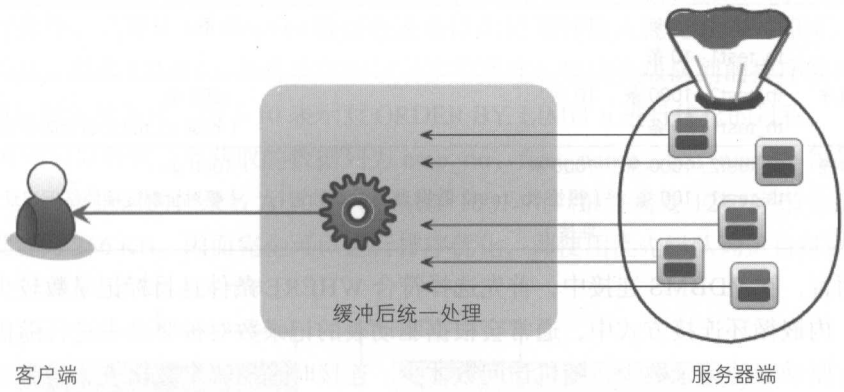


图 5-9 缓冲处理

前面提到过，ORDER BY 的 3 种处理方式中，只有“使用索引排序方式”采用流处理，其他处理方式均需要先缓冲再进行排序处理。也就是说，使用索引排序的方式下，只读取 LIMIT 指定的记录数，并将结果立即发送给客户端。但对于无法使用索引的排序处理，需要先从磁盘读取所有需要的记录，完成排序后才能根据 LIMIT 条件选取指定的记录条数并传送给客户端。

查询涉及连接且同时含有 ORDER BY 与 LIMIT 子句时，各种排序处理方式有何差别呢？下面进行详细分析。

```
SELECT *
FROM tb_test1 t1, tb_test2 t2
```

```
WHERE t1.col1=t2.col1
ORDER BY t1.col2
LIMIT 10;
```

假设 tb_test1 数据表的记录为 100 条，tb_test2 数据表的记录为 1000 条（tb_test1 的 1 条记录对应 tb_test2 的 10 条记录），两个数据表的连接结果总共 1000 条。下面针对不同排序处理方式下要读取和排序的记录数进行比较。

tb_test1 为驱动表时

排序方式	要读取的记录数	访问次数	要排序的记录数
使用索引	tb_test1：1 条 tb_test2：10 条	1 次	0 条
只对驱动表排序	tb_test1：100 条 tb_test2：10 条	10 次	100 条 （根据 tb_test1 数据表的记录数进行排序）
使用临时表排序	tb_test1：100 条 tb_test2：1000 条	100 次 （根据 tb_test1 数据表的记录数进行连接）	1000 条 （要对全部连接的结果记录条数进行排序）

tb_test2 为驱动表时

排序方式	要读取的记录数	访问次数	要排序的记录数
使用索引	tb_test2：10 条 tb_test1：10 条	10 次	0 条
只对驱动表排序	tb_test2：1000 条 tb_test1：10 条	10 次	1000 条 （根据 tb_test2 数据表的记录数进行排序）
使用临时表排序	tb_test2：1000 条 tb_test1：100 条	1000 次 （根据 tb_test2 数据表的记录数进行连接）	1000 条 （要对全部连接的结果记录条数进行排序）

一般而言，在 RDBMS 连接中，首先选择符合 WHERE 条件且目标记录数较少的数据表作为驱动表。内嵌循环连接方式中，通常会根据驱动表的记录数对被驱动表进行随机访问。也就是说，只有驱动表的记录数少，随机访问数才少。连接时选择哪个数据表作为驱动表是很重要的，当然，所选的排序处理方式也会对查询性能产生很大影响。查询调优时，若有可能，请尽量使用索引进行排序；若不能，请选择只对驱动表进行排序的方式。

注意 无法使用索引又需要另外进行 Filesort 处理的查询中，LIMIT 条件也不是毫无用处。若需要排序的记录有 1000 条，且查询中存在 LIMIT 10 限制条件，则 MariaDB 服务器在处理时不会对 1000 条记录全部进行排序，而会根据需要的顺序（ASC 或 DESC）排序后，只对前面 10 条记录进行排序，然后停止排序并返回结果。MariaDB 服务器采用快速排序算法，这意味着，对 LIMIT 10 限制的前面 10 条记录排序时，可能需要做更多处理工作。快速排序算法相关内容可以从网上轻松查找，请各位自学。

总而言之，分页处理中采用无法使用索引的查询时，即使通过 LIMIT 子句将访问记录条数限制为 5~10 条，查询也不会变得像期望的那样快。

5.2.4 ORDER BY..LIMIT n 优化 MySQL

MySQL 与 MariaDB 中，带有 ORDER BY 子句的查询无法使用索引时，MariaDB 服务器就会只进行实时排序（Using filesort）。此时会读取所有符合条件的记录，利用排序缓冲运行快速排序算法（Quick sort）。若要排序的目标记录非常多，会根据排序缓冲大小获取记录进行排序，然后再与已经排好序的结果进行合并（Sort_merge_pass）。整个过程对 CPU 和磁盘都会带来很大负担。

MySQL 5.6 与 MariaDB 10.0 中，对同时带有 ORDER BY 与 LIMIT n 子句的查询进行了改善，采用略有不同的方式进行优化。假如有如下查询，它无法使用索引进行排序。该查询先从 employees 数据表中查找符合 WHERE 条件的 900 条雇员记录，然后采用快速排序算法以 last_name 列为基准进行排序。

```
SELECT emp_no, first_name, last_name FROM employees
WHERE emp_no BETWEEN 10001 AND 10900
ORDER BY last_name
LIMIT 10;
```

上述查询中，需要排序的记录数为 900 条，但查询最终只返回 10 条记录。要排序的记录很多而最终返回的记录较少时，MySQL 5.6 会在排序缓冲中创建优先级队列（Priority Queue），然后利用队列排序。像这样，如果使用优先级队列，就没有必要使用快速排序算法对 900 条结果记录全部进行排序，只要从 employees 数据表逐条读出记录并放入优先级队列即可。该处理过程中最重要的是，根据 LIMIT n 条件获取的记录字节数比创建优先级队列的排序缓冲小，这样才能应用上面这种优化方式。图 5-10 表示对 ORDER BY..LIMIT n 进行优化的过程。

从图 5-10 中可以看到，要获取的数据列为 emp_no、first_name、last_name，每条记录最大为 124 字节。又因要返回的记录数为 10 条（LIMIT 10），所以最终需要 1240 字节空间。而排序缓冲大小被设置为 256 KB，因而完全可以进行排序优化，即使用优先级队列进行排序。

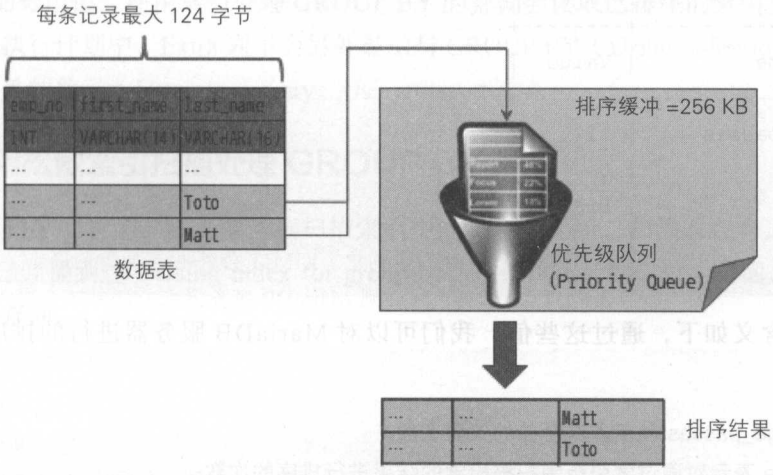


图 5-10 使用优先级队列进行排序优化

像这样，使用优先级队列排序可以不必使用复杂的快速排序算法，而且排序目标很大时，也不必对所需部分进行排序与合并。遗憾的是，目前的MySQL版本中，还无法从执行计划判断排序时究竟使用了优先级队列方式还是快速排序算法，两者执行计划的Extra列都显示为Using filesort。

注意 MariaDB还不支持使用优先级队列排序，目前正在实现该功能。MariaDB开发网站上明确指出，将向MariaDB 10.0版本添加该功能。

5.2.5 与排序相关的状态变量

进行重要处理时，MariaDB服务器会将相关处理的执行次数存储到相关状态变量。进行排序处理时，可以使用下列命令查看当前已排序的记录数以及排序缓冲间的合并（Multi-merge）处理次数。

```
MariaDB> SHOW SESSION STATUS LIKE 'Sort%';
+----------------------+--------+
|Variable name         | Value  |
+----------------------+--------+
|Sort_merge_passes     | 56     |
|Sort_range            | 0      |
|Sort_rows             | 279408 |
|Sort_scan             | 1      |
+----------------------+--------+
MariaDB> SELECT first_name, last_name
       FROM employees
       GROUP BY first_name, last_name;
MariaDB> SHOW SESSION STATUS LIKE 'Sort%';
+----------------------+--------+
|Variable name         | Value  |
+----------------------+--------+
|Sort_merge_passes     | 112    |
|Sort_range            | 0      |
|Sort_rows             | 558816 |
|Sort_scan             | 2      |
+----------------------+--------+
```

各状态值含义如下，通过这些值，我们可以对MariaDB服务器进行的排序处理有深入了解。

- Sort_merge_passes表示多次合并的处理次数。
- Sort_range表示对通过索引范围扫描检索的结果进行排序的次数。
- Sort_scan表示对通过全表扫描检索的结果进行排序的次数。Sort_range与Sort_scan均为排序处

理次数的累积值。

- Sort_rows 表示目前为止已排序的全部记录数。

分析上述示例结果可知：

(1) 对全表扫描结果进行 1 次（2-1=1）排序；

(2) 对单位排序结果进行 56 次（112-56）合并处理；

(3) 总排序记录数为 279 408 条（558 816-279 408=279 408）。

5.3 GROUP BY 处理 MariaDB MySQL

与 ORDER BY 一样，GROUP BY 也是使查询无法被流处理的因素。带有 GROUP BY 子句的查询中，可以使用 HAVING 子句，它用于对 GROUP BY 的结果进行过滤处理。GROUP BY 的处理结果一般负责对存在于临时表或缓冲的值进行过滤。处理 GROUP BY 中的条件时，无法使用索引，所以优化 HAVING 子句时，不必创建索引或考虑使用其他方法。

GROUP BY 处理也分为可使用索引与不可使用索引两种情形。可使用索引的情形下，又有索引扫描（依次使用索引）与松散索引扫描（跳跃索引扫描）两种方法。而不可使用索引的查询中，要使用临时表处理 GROUP BY。

5.3.1 使用索引扫描处理 GROUP BY（紧凑索引扫描）

与 ORDER BY 一样，仅使用连接驱动表中的数据列进行分组时，若 GROUP BY 的参照列（分组依据的列）已经创建索引，则依次读取索引，同时进行分组处理，然后再对分组结果进行连接处理。即使采用索引处理 GROUP BY，有时也需要临时表处理聚合函数（Aggregation function）等的分组值。使用索引处理 GROUP BY 的查询会读取已排序的索引。使用这种分组方式的查询的执行计划中，Extra 列不会另外显示与 GROUP BY（Using index for group-by）、临时表或排序相关的信息（Using temporary、Using filesort）。

5.3.2 使用松散索引扫描处理 GROUP BY

松散索引扫描方式下，不需要连续扫描索引中的每一个元组，扫描时仅考虑索引中的一部分。4.3.10 节已经提到过“Using index for group-by”这一执行计划信息，下面介绍使用松散索引扫描的示例查询。

```
EXPLAIN
SELECT emp_no
FROM salaries
WHERE from_date='1985-03-01'
GROUP BY emp_no;
```

由于 salaries 数据表的索引为 (emp_no+from_date)，所以上述查询语句的 WHERE 条件无法使用索引范围扫描访问方式。但从查询的执行计划看，它使用了索引范围扫描（range 类型），并且 Extra 列也显示了使用索引处理 GROUP BY 的信息，如下所示。

id	select_type	Table	type	key	key_len	ref	rows	Extra
1	SIMPLE	salaries	range	PRIMARY	7		568914	Using where; Using index for group-by

MariaDB 服务器执行示例查询的步骤如下所示：

(1) 依次扫描 (emp_no+from_date) 索引，找出 emp_no 的首个唯一值（组键）“10001”；

(2) (emp_no+from_date) 索引中，从 emp_no 为 10001 记录只获取 from_date 值为 “1985-03-01” 的记录。该检索方法将步骤 1 中找到的 “10001” 值与查询的 WHERE 子句中使用的 “from-date='1985-03-01'” 条件进行合并，类似于使用 “emp_no=10001 AND from-date='1985-03-01'” 条件检索 (emp_no+from_date) 索引；

(3) (emp_no+from_date) 索引中，获取 emp_no 的下一个唯一值（组键）；

(4) 步骤 3 中无更多结果时终止处理，否则返回步骤 2 重复执行。

MariaDB 服务器只在对单一数据表进行 GROUP BY 处理时才使用松散索引扫描方式，且前缀索引（Prefix index，仅用列值的前面一部分创建的索引）无法使用松散索引扫描方式。索引范围扫描中，唯一值的数量越多，性能越好；而松散索引扫描中，索引的唯一值越少，性能越好。也就是说，对于分布图越不好的索引，使用松散索引扫描方式越能快速产生结果。使用松散索引扫描处理的查询中，不必使用另外的临时表。

判断是否可用松散索引扫描比判断 WHERE 子句条件或 ORDER BY 子句是否可以使用索引更加困难。下面看一些不同类型的查询，并进行判断，判断其是否可以使用松散索引扫描。首先，假设 tb_test 数据表的索引为 (col1+col2+col3)。下列查询均可使用松散索引扫描。仔细观察这些查询类型，想想为何它们可以使用松散索引扫描。

```
SELECT col1, col2 FROM tb_test GROUP BY col1, col2;
SELECT DISTINCT col1, col2 FROM tb_test;
SELECT col1, MIN(col2) FROM tb_test GROUP BY col1;
SELECT col1, col2 FROM tb_test WHERE col1 < const GROUP BY col1, col2;
SELECT MAX(col3), MIN(col3), col1, col2 FROM tb_test WHERE col2 > const
GROUP BY col1, col2;
SELECT col2 FROM tb_test WHERE col1 < const GROUP BY col1, col2;
SELECT col1, col2 FROM tb_test WHERE col3 = const GROUP BY col1, col2;
```

下列查询均不可使用松散索引扫描。

```
--// 因使用集合函数（MIN()、MAX() 除外）而无法使用松散索引扫描。
SELECT col1, SUM(col2) FROM tb_test GROUP BY col1;
```

```
--// 因 GROUP BY 中的数据列从索引组成数据列的左侧起不一致而无法使用松散索引扫描。
SELECT col1, col2 FROM tb_test GROUP BY col2, col3;

--// 因 SELECT 中的数据列与 GROUP BY 中的不一致而无法使用松散索引扫描。
SELECT col1, col3 FROM tb_test GROUP BY col1, col2;
```

注意 通常，只有组成索引的数据列以左起不一致的形态被使用时，才可以使用 B-Tree 索引。但即使索引的第一个数据列未被使用，在 WHERE 条件或 GROUP BY 中，松散索引扫描方式也可以使用 B-Tree 索引。为实现该功能，在 Oracle 这样的 DBMS 中，优化器可以随意创建并添加索引的第一个数据列的条件。但是 MariaDB 与 MySQL 仍未提供这样使用松散索引扫描的方法。

5.3.3 使用临时表处理 GROUP BY

无论 GROUP BY 的参照列（分组依据的列）属于驱动表还是被驱动表，只要无法使用索引，就会使用临时表处理 GROUP BY。请看下列示例查询。

```
EXPLAIN
SELECT e.last_name, AVG(s.salary)
FROM employees e, salaries s
WHERE s.emp_no=e.emp_no
GROUP BY e.last_name;
```

上述查询的执行计划中，Extra 列显示“Using temporary”与“Using filesort”信息。该查询的执行计划中，使用临时表不是因为对 employees 数据表进行了全扫描（ALL），而是因为无法使用索引处理 GROUP BY 子句。

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	e	ALL				300024	Using temporary; Using filesort
1	SIMPLE	s	ref	PRIMARY	4	e.emp_no	9	

图 5-11 描述了该执行计划的处理过程。

(1) 使用全表扫描方式读取 employees 数据表；

(2) 使用步骤 1 中读取的 employees 数据表的 emp_no 值检索 salaries 数据表；

(3) 将步骤 2 中获取的连接结果记录保存到临时表。此时临时表仅存储着原查询中 GROUP BY 子句的数据列与查询（SELECT）的数据列。临时表中使用 GROUP BY 子句中的数据列创建唯一键，即使用临时表处理 GROUP BY 时，所用的临时表总是拥有唯一键；

(4) 重复执行步骤 1 到步骤 3，直到连接完成，之后依据临时表的唯一键顺序读取记录并返回给客户。若 ORDER BY 子句中的数据列与 GROUP BY 子句中的数据列一样，则不另外进行排序处理；否则要经过 Filesort 过程，再次进行排序处理。

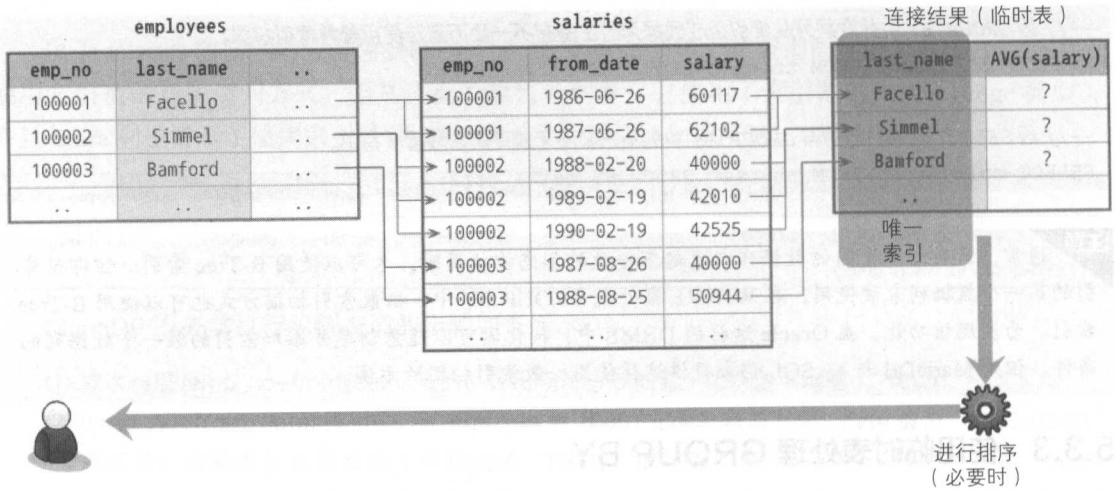


图 5-11 使用临时表处理 GROUP BY

5.4 DISTINCT 处理 MariaDB MySQL

若只想访问特定数据列的唯一值，则要在 SELECT 查询中使用 DISTINCT 关键字。根据 DISTINCT 关键字的作用范围不同，使用 DISTINCT 可分为与集合函数（MIN()、MAX()、COUNT() 等）并用以及不使用集合函数的情形。此外，DISTINCT 与集合函数并用的查询的执行计划中，DISTINCT 处理无法使用索引时，总是使用临时表，但执行计划的 Extra 列不会显示 Using temporary 信息。

5.4.1 SELECT DISTINCT...

若只想从查询（SELECT）到的记录中获取唯一记录（不重复记录），则编写查询语句时就要使用 SELECT DISTINCT 语句。此时使用类似 GROUP BY 的处理方式，不同之处在于，SELECT DISTINCT 无法保证排列顺序。请看下列两个查询，它们除了排序相关部分外，其他部分采用相同的内部处理。然而，由于它们全部使用索引，所以不需要额外进行排序处理，是完全相同的查询。但 DISTINCT 无法使用索引时，则无法保证所得记录的排列顺序。

```
SELECT DISTINCT emp_no FROM salaries;
SELECT emp_no FROM salaries GROUP BY emp_no;
```

我们使用 DISTINCT 时经常犯错。执行 DISTINCT 时，会从查询（SELECT）到的记录（元组）中选择唯一记录，而不是查询唯一的数据列。下列示例查询中，最终的查询结果不是获取只有 first_name 数据列唯一的记录，而是获取 first_name 与 last_name 两个数据列都唯一的记录。

```
SELECT DISTINCT first_name, last_name FROM employees;
```

有时也如下使用 DISTINCT。

```
SELECT DISTINCT(first_name), last_name FROM employees;
```

乍一看，上述查询只查询 first_name 列唯一的记录，查询 last_name 列时似乎不受 DISTINCT 限制，就像不使用 DISTINCT 关键字一样。而且，实际执行上述查询时，也不会发生任何错误，所以很容易引起用户的误解。请注意，MariaDB 服务器会将 DISTINCT 后面的括号视为无意义的括号，并将其删除。由于 DISTINCT 不是函数，所以它后面的括号没有任何意义。

```
SELECT DISTINCT first_name, last_name FROM employees;
```

SELECT 子句中使用的 DISTINCT 关键字会影响所有查询的数据列，不可能只影响所查询数据列的一部分。但 DISTINCT 用于集合函数内部时，其作用范围就不同了。

5.4.2 DISTINCT 用于集合函数内部

DISTINCT 关键字也可用于 COUNT()、MIN()、MAX() 这类集合函数的内部，此时，其作用范围与以 SELECT DISTINCT 形式使用时的作用范围不同。DISTINCT 用在不含集合函数的 SELECT 查询中时，它作用于要查询的所有列，即获取查询的所有列都唯一的记录。而用于集合函数内的 DISTINCT 则只作用于以函数参数形式传递的数据列，即让参数数据列的值唯一。

```
EXPLAIN
SELECT COUNT(DISTINCT s.salary)
FROM employees e, salaries s
WHERE e.emp_no=s.emp_no
AND e.emp_no BETWEEN 100001 AND 100100;
```

上述查询在内部使用临时表处理“COUNT(DISTINCT s.salary)”。但该查询的执行计划中，并不显示使用临时表的相关信息。这看上去像 BUG，但在 MariaDB 与 MySQL 所有版本的执行计划中，Extra 列都不会显示 Using temporary 信息。

id	select_type	table	type	key	key_len	Ref	rows	Extra
1	SIMPLE	e	range	PRIMARY	4		100	Using where; Using index
1	SIMPLE	s	ref	PRIMARY	4	e.emp_no	9	

就上述查询而言，employees 与 salaries 数据表的连接结果中，创建并使用临时表以存储 salary 的列值。此时，由于临时表的 salary 数据列创建有唯一索引，所以记录数增多时，查询会变得相当缓慢。

接下来，在上述查询中再添加一个 COUNT(DISTINCT...)，形成如下查询。COUNT() 函数使用 2 次之后，查询的执行计划与上述查询的完全一样。但处理下列查询时，总共需要使用 2

个临时表，一个存储 s.salary 列值，另一个存储 e.last_name 列值。

```
SELECT COUNT(DISTINCT s.salary), COUNT(DISTINCT e.last_name)
FROM employees e, salaries s
WHERE e.emp_no=s.emp_no
AND e.emp_no BETWEEN 100001 AND 100100;
```

上述查询中，由于处理 DISTINCT 时无法使用索引，因而需要使用临时表。但下列查询中，对建有索引的数据列进行 DISTINCT 处理时，使用索引全扫描或范围扫描，执行优化后的处理则不需要使用临时表。

```
SELECT COUNT(DISTINCT emp_no) FROM employees;
SELECT COUNT(DISTINCT emp_no) FROM dept_emp GROUP BY dept_no;
```

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	dept_emp	range	PRIMARY	16		331603	Using index

注意 如前所述，DISTINCT 关键字可以用于不含集合函数的查询，也可用于集合函数内部。这两种情形下的查询结果略有不同，各位要准确理解。请比较并牢记下列 3 个查询的不同之处。

```
SELECT DISTINCT first_name, last_name
FROM employees
WHERE emp_no BETWEEN 10001 AND 10200;

SELECT COUNT(DISTINCT first_name), COUNT(DISTINCT last_name)
FROM employees
WHERE emp_no BETWEEN 10001 AND 10200;

SELECT COUNT(DISTINCT first_name, last_name)
FROM employees
WHERE emp_no BETWEEN 10001 AND 10200;
```

5.5 临时表（Using temporary） MariaDB MySQL

对从存储引擎接收的记录进行排序或分组时，MariaDB 引擎会使用内部临时表。此处之所以使用“内部”一词，是为了将其与使用 CREATE TEMPORARY TABLE 语句创建的临时表进行区分。通常，MariaDB 引擎使用的临时表最初是在内存中创建的，然后随着表的增大而转移到磁盘。当然，某些特定情形下，也可以不使用内存而直接在磁盘上创建临时表。不受原数据表的存储引擎影响的前提下，临时表使用内存时会使用 MEMORY 存储引擎，保存到磁盘时会使用 Aria 存储引擎。

注意 与 MariaDB 不同，MySQL 与 Percona Server 需要将内部临时表存储到磁盘时，会使用 MyISAM 存储引擎。因此，在 MySQL 中需要对 MyISAM 进行优化设置。MariaDB 中，mysql 数据库的几个数据表仍然使用 MyISAM 存储引擎，所以需要对 Aria 存储引擎和 MyISAM 存储引擎进行优化设置。

MariaDB 为内部数据加工而创建的临时表在其他会话或查询中都是不可见的，并且也无法使用。与用户创建的临时表（CREATE TEMPORARY TABLE）不同，内部临时表在查询处理结束后会自动删除。

5.5.1 需要使用临时表的查询

对于下列类型的查询，MariaDB 引擎处理时会创建内部临时表，以便另外对数据进行加工处理。此外，无法使用索引时，很多时候也要创建临时表以处理查询。

- 查询中 ORDER BY 与 GROUP BY 中的数据列不同。
- 查询中 ORDER BY 或 GROUP BY 中的数据列不在第一个（连接顺序）数据表。
- 查询中同时有 DISTINCT 与 ORDER BY 时，或无法使用索引处理 DISTINCT 时。
- 查询使用 UNION 或 UNION DISTINCT 时（select_type 列为 UNION RESULT 时）。
- 查询使用 UNION ALL 时（select_type 列为 UNION RESULT 时）。
- 查询的执行计划中，select_type 列为 DERIVED 时。

根据查询的执行计划的 Extra 列是否显示 Using temporary 信息，可以判断处理查询时是否使用临时表。但有时即使执行计划的 Extra 列不显示 Using temporary，处理查询时也会使用临时表，比如上述最后 3 种查询。第一种到第四种查询中，处理时会创建拥有唯一索引的内部临时表；而处理第五种与第六种查询时，创建没有唯一索引的内部临时表。一般而言，处理查询时，使用带有唯一索引的内部临时表要比使用不带唯一索引的内部表的处理速度要慢很多。

5.5.2 在磁盘上创建临时表（使用 Aria 存储引擎） MariaDB

内部临时表通常是在内存中创建的，但下列情形下，临时表由磁盘上使用 Aria 存储引擎的数据表创建而成。

- 要存储在临时表中的内容包含 BLOB 或 TEXT 类型的大容量数据列时。
- 要存储在临时表中的记录的全部大小或从 UNION、UNION ALL 中查询（SELECT）到的列中存在长度大于 512 字节的数据列时。
- GROUP BY 或 DISTINCT 数据列中含有 512 字节以上的数据列。
- 要存储在临时表中的数据的全部大小（数据的字节大小）比 tmp_table_size 或 max_heap_table_size 系统设置值大时。

对于第一种到第三种情况，一开始就使用 Aria 存储引擎在磁盘上创建内部临时表。而对于第四种情况，起初使用 MEMORY 存储引擎在内存中创建内部临时表，临时表的大小超过系统设置值后，即被转换为磁盘上的 Aria 数据表。

5.5.3 与临时表有关的状态变量

执行计划的 Extra 列显示 Using temporary 信息时，表示处理查询时要使用临时表。但无法得知临时表在内存中还是在磁盘上，也不知道使用了几个临时表。并不是显示 1 次 Using temporary 就意味着只使用了 1 个临时表。若想知道临时表是在磁盘上创建的还是在内存中生成的，可以通过 MariaDB 服务器的状态变量（SHOW SESSION STATUS LIKE'Created_tmp%';）进行查看。

```
MariaDB> SHOW SESSION STATUS LIKE 'Created_tmp%tables';
+-------------------------+--------+
|Variable name            | Value  |
+-------------------------+--------+
|Created_tmp_disk_tables | 0      |
|Created_tmp_tables       | 0      |
+-------------------------+--------+
MariaDB> SELECT first_name, last_name

FROM employees

GROUP BY first_name, last_name;

MariaDB> SHOW SESSION STATUS LIKE 'Created_tmp%tables';
+-------------------------+--------+
|Variable name            | Value  |
+-------------------------+--------+
|Created_tmp_disk_tables | 1      |
|Created_tmp_tables       | 1      |
+-------------------------+--------+
```

如上所示，执行查询前，先使用“SHOW SESSION STATUS LIKE'Created_tmp%';”命令查看临时表的使用情况。然后执行 SELECT 查询，再次使用状态查看命令进行查看。上述示例查询中，各状态变量含义如下所示。

Created_tmp_tables

该变量值为处理查询时创建的内部临时表的数量。它是累加值，所统计的内部临时表既包含在内存中创建的，也包含在磁盘上创建的。

Created_tmp_disk_tables

该变量只统计磁盘上创建的临时表的数量。

从示例查询内部临时表的使用情况看，总共创建了 1 个（3–2=1）临时表；又因 Created_tmp_disk_tables 变量值为 1，由此可知临时表是在磁盘上创建的。

5.5.4 带索引的内部临时表

MySQL 5.5 与 MariaDB 5.2 之前，内部临时表都是不带索引的数据表。因此，连接中将内部临时表（Derived table）用作被驱动表（非驱动表）或查询表（Lookup table）时，性能会变得非常低下。但从 MariaDB 5.3 开始，MariaDB 引擎会分析查询的特征，自动生成带索引的临时表。

下列示例查询中，先依据 dept_name 数据列对 departments 数据表进行分组，然后创建两个临时表（x 与 y）保存分组结果，再使用两个表的别名进行连接。由于两个表都要进行临时表间的连接，而 MariaDB 5.2 与 MySQL 5.5 中无法使用索引，所以每次都要使用全表扫描检索临时表。

```
MariaDB> EXPLAIN SELECT *
         FROM
           (SELECT dept_name FROM departments GROUP BY dept_name) x,
           (SELECT dept_name FROM departments GROUP BY dept_name) y
         WHERE x.dept_name=y.dept_name;
```

MariaDB 10.0 中，上述示例查询的执行计划如下所示。

id	select_type	table	type	Key	key_len	ref	rows	Extra
1	PRIMARY	<derived2>	ALL	NULL	NULL	NULL	9	
1	PRIMARY	<derived3>	ref	key0	123	x.dept_name	2	
3	DERIVED	departments	index	ux_deptname	122	NULL	9	Using index
2	DERIVED	departments	index	ux_deptname	122	NULL	9	Using index

上述执行计划中，显示为 <derived2> 的内部临时表（“x”）为驱动表，而显示为 <derived3> 的内部临时表（“y”）为被驱动表，它们正在进行连接操作。该执行计划中需要注意的是，第二行的 key 字段显示为 key0，表示临时表 y 的 dept_name 数据列上创建了索引。

MariaDB 10.0 中，默认优化器选项设置为内部临时表自动创建索引，可以通过 optimizer_switch 变量进行确认。若当前处于关闭状态，可以使用如下命令开启临时表键的创建功能。

```
MariaDB> SET optimizer_switch='derived_with_keys=on';
```

5.5.5 内部临时表的注意事项

若记录数不多，可以直接在内存中创建内部临时表，而不会对 MariaDB 服务器的负荷产生很大影响。而在磁盘上创建内部临时表（Aria 数据表）时，可能出现一些性能问题。

```
SELECT * FROM employees GROUP BY last_name ORDER BY first_name;
```

上述查询中，GROUP BY 与 ORDER BY 中的数据列是不同的。由于 last_name 数据列无索引，所以执行查询时需要使用临时表进行分组和排序处理。

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	employees	ALL				300024	Using temporary; Using filesort

从执行计划的 Rows 列值可知，执行该查询要处理的记录数约为 30 万条。以上执行计划的内部处理过程如下：

(1) 创建临时表（MEMORY 数据表），用于存储 employees 数据表中的所有数据列；
(2) 从 InnoDB 存储引擎获取 employees 数据表的第一条记录；
(3) 检查临时表中是否存在相同的 last_name；
(4) 若无相同的 last_name，则插入（INSERT）到临时表；
(5) 若有相同的 last_name，则更新（UPDATE）到临时表或者忽略；
(6) 临时表大小超过指定大小时，借助 Aria 存储引擎将临时表转移到磁盘；
(7) 重复步骤 2~6，直到 employees 表中再无可读记录（本示例查询中要重复 30 万次）；
(8) 对存储在内部临时表中的结果进行排序处理；
(9) 将结果返回给客户。

根据 SELECT 子句所含数据列的特征，步骤 3~5 过程略有不同，但临时表一般都会经过以上过程。临时表位于内存时，问题不大；但存储于磁盘中时，读写 30 万条记录会对系统产生不小的负荷。若有可能，建议使用索引处理，这样从一开始就不需要使用临时表。若不能使用索引，则减少要加工的目标记录的数量，使它们可以存储在位于内存中的内部临时表。但无法减少要加工的数据时，若将 tmp_table_size 或 max_heap_table_size 系统变量设置增大，MariaDB 服务器要使用的空闲内存都会被临时表用掉，请注意这一点。

在物理内存中使用 MEMORY(HEAP) 数据表创建临时表时，要注意 MEMORY(HEAP) 数据表的所有列都是定长的。假设上述示例查询中，first_name 数据列为 VARCHAR(512)。实际物理数据表中，first_name 数据列占据的空间为 512*3（假定字符集为 utf8）字节。无论实际 first_name 列值为 1 个字还是 2 个字，在物理数据表中占据的空间与定义数据表时指定的大小一样。临时表的这种存储方式导致查询的数据列要最小化（若非必要，不要含 BLOB 或 TEXT 列），为数据列选择数据类型时，尽可能选择小的。

注意 MariaDB 的使用手册中指出，MariaDB 10.0 的 MEMORY 存储引擎支持可变长字符串类型（VARCHAR），但事实并非如此。因此，在 MariaDB 10.0 中需要使用 VARCHAR 类型时，内部仍然使用定长的 CHAR 类型而非 VARCHAR。

5.6 索引条件下推 MariaDB MySQL

MariaDB 5.3 与 MySQL 5.6 开始引入了“索引条件下推”（Index Condition Pushdown，ICP）功能。其实，“索引条件下推”并非新技术，而是早已出现。只是当时效率非常低，后来经过不断改善发展到现在，成为比较成熟的技术。测试之前先创建索引，然后调整 optimizer_switch，

关闭“索引条件下推”功能。

```
MariaDB> ALTER TABLE employees ADD INDEX ix_lastname_firstname (last_name, first_name);
MariaDB> SET optimizer_switch='index_condition_pushdown=off';
MariaDB> SHOW VARIABLES LIKE 'optimizer_switch';
Variable_name: optimizer_switch
        Value: ...,index_condition_pushdown=off,...
```

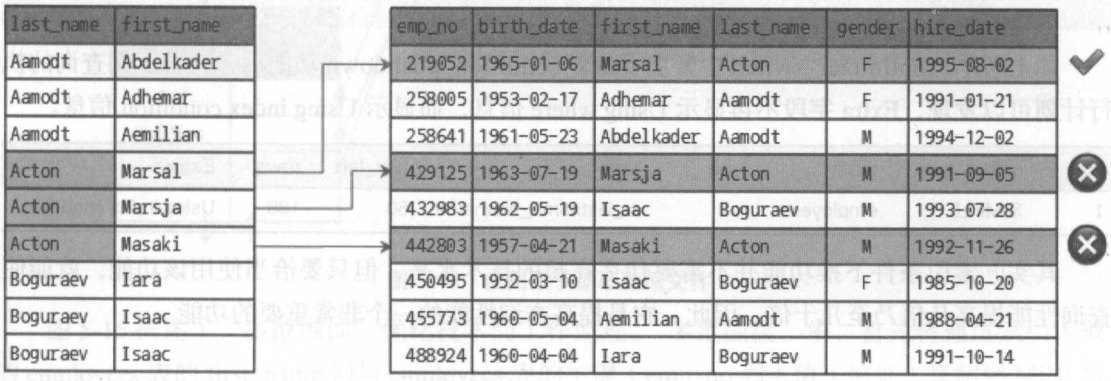
执行如下查询，观察存储引擎读取了多少条记录。下述示例查询中，处理 last_name='Acton' 条件时，通过范围扫描使用上面创建的 ix_lastname_firstname 索引。但 first_name LIKE'%sal' 条件使用索引范围扫描无法缩小要检索的索引范围，所以下述查询中，last_name 条件只能访问 ix_lastname_firstname 索引的特定范围，而 first_name LIKE'%sal' 条件仅用作检测条件（也称为过滤条件）。处理时先要读取所有数据，然后逐条比较检查是否为用户需要的结果。

```
MairaDB> SELECT * FROM employees WHERE last_name='Action' AND first_name LIKE '%sal';
```

上述查询的执行计划如下所示，Extra 列显示 Using where 信息，表示对存储引擎（如 InnoDB/XtraDB）读取并返回的记录进行检测，检测其是否符合无法使用索引的 WHERE 条件。该查询中，first_name LIKE'%sal' 条件用于检测过程。

id	select_type	Table	type	key	key_len	rows	Extra
1	SIMPLE	employees	ref	ix_lastname_firstname	50	188	Using where

图 5-12 表示整个比较过程。先用 last_name='Acton' 条件进行索引范围扫描，读取数据表记录，然后进行比较，检查是否符合 first_name LIKE'%sal' 条件。虽然看上去与性能关系不大，但上述过程具有非常重要的意义。如图 5-12 所示，虽然实际从数据表中读取了 3 条记录，但其实只有 1 条符合 first_name LIKE'%sal' 条件。若满足 last_name='Acton' 条件的记录约有 10 万条，而其中只有 1 条满足 first_name LIKE'%sal' 条件，会怎么样呢？此时，不必对 99 999 条记录进行读取操作。



last_name	first_name
Aamodt	Abdelkader
Aamodt	Adhemar
Aamodt	Aemilian
Acton	Marsal
Acton	Marsja
Acton	Masaki
Boguraev	Iara
Boguraev	Isaac
Boguraev	Isaac

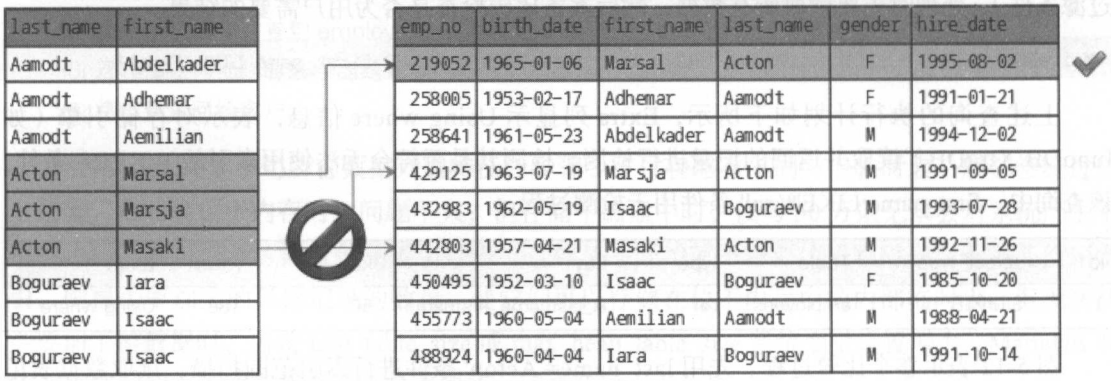
emp_no	birth_date	first_name	last_name	gender	hire_date
219052	1965-01-06	Marsal	Acton	F	1995-08-02
258005	1953-02-17	Adhemar	Aamodt	F	1991-01-21
258641	1961-05-23	Abdelkader	Aamodt	M	1994-12-02
429125	1963-07-19	Marsja	Acton	M	1991-09-05
432983	1962-05-19	Isaac	Boguraev	M	1993-07-28
442803	1957-04-21	Masaki	Acton	M	1992-11-26
450495	1952-03-10	Isaac	Boguraev	F	1985-10-20
455773	1960-05-04	Aemilian	Aamodt	M	1988-04-21
488924	1960-04-04	Iara	Boguraev	M	1991-10-14

图 5-12 关闭索引条件下推功能时

再仔细想想就会产生如下疑问：处理 first_name LIKE'%sal' 条件时，是否没有使用已经读取

的 ix_lastname_firstname 索引的 first_name 数据列，而又再次读取了数据表中的记录呢？若使用索引的 first_name 列进行比较，就可以不用再从数据表中读取不必要的 2 条记录。其实，使用索引中的 first_name 列还是使用数据表中的 first_name 列，是由 first_name LIKE'%sal' 条件的处理者决定的。图 5-12 中，索引比较是由 InnoDB/XtraDB 存储引擎进行的；而数据表的记录中，比较 first_name 条件是由 MySQL 或 MariaDB 引擎进行的。MySQL 5.5 或 MariaDB 5.2 之前的版本中，first_name 条件不能用作索引范围限制条件时，MySQL 或 MariaDB 引擎根本不会将其传递给存储引擎。因此，存储引擎最终还是要从数据表读取不必要的 2 条记录。

MariaDB 5.3 与 MySQL 5.6 开始，改善了 Handler API。即使索引无法被用作范围限制条件，包含在索引中的数据列条件都会一起被传递给存储引擎。因此，MariaDB 5.3 与 MySQL 5.6 开始，使用索引最大限度地完成过滤后，将只从数据表中读取所需的 1 条记录，如图 5-13 所示。



last_name	first_name
Aamodt	Abdelkader
Aamodt	Adhemar
Aamodt	Aemilian
Acton	Marsal
Acton	Marsja
Acton	Masaki
Boguraev	Iara
Boguraev	Isaac
Boguraev	Isaac

emp_no	birth_date	first_name	last_name	gender	hire_date
219052	1965-01-06	Marsal	Acton	F	1995-08-02
258005	1953-02-17	Adhemar	Aamodt	F	1991-01-21
258641	1961-05-23	Abdelkader	Aamodt	M	1994-12-02
429125	1963-07-19	Marsja	Acton	M	1991-09-05
432983	1962-05-19	Isaac	Boguraev	M	1993-07-28
442803	1957-04-21	Masaki	Acton	M	1992-11-26
450495	1952-03-10	Isaac	Boguraev	F	1985-10-20
455773	1960-05-04	Aemilian	Aamodt	M	1988-04-21
488924	1960-04-04	Iara	Boguraev	M	1991-10-14

图 5-13 索引条件下推

下面将优化器选项恢复为原设置，再次查看示例查询的执行计划。

```
MariaDB> SET optimizer_switch='index_condition_pushdown=on';
MariaDB> SHOW VARIABLES LIKE 'optimizer_switch';
Variable_name: optimizer_switch
        Value: ...,index_condition_pushdown=on,...
```

如上所示，optimizer_switch 中开启 index_condition_pushdown 功能后，查看示例查询的执行计划可以发现，Extra 字段不再显示 Using where 信息，而显示 Using index condition 信息。

id	select_type	table	type	key	key_len	rows	Extra
1	SIMPLE	employees	ref	ix_lastname_firstname	50	188	Using index condition

其实，索引条件下推功能并不需要什么高超的技术水平，但只要恰当使用该功能，就能使查询性能提高几倍乃至几十倍。因此，它是提高查询性能的一个非常重要的功能。

5.7 多范围读 MariaDB MySQL

传统 RDBMS 通过索引检索快速选择目标记录范围，而通过索引读取实际数据文件过程中会发生随机 I/O，有多少条记录就发生多少次随机 I/O。图 5-14 中，先使用创建在 employees 数据表的 first_name 列上的索引进行检索，然后通过随机 I/O 方式读取数据文件，以获取记录其余的数据列。

为了减少磁盘随机 I/O，MariaDB 引入了“多范围读”（Multi range read，MRR）优化技术。令人意外的是，“多范围读”优化算法很简单。它并不是先读取索引，再直接进行随机 I/O 以读取数据文件中的数据；而是先读取合适数量的索引元组，再按照 RowId 顺序对元组排序，然后读取数据文件中的记录。RowId 指向数据文件中记录的存储位置，按照 RowId 排序后，读取数据文件中的记录时，只要顺序读取即可。进行顺序读取时，存储在同一数据块（页面）的记录会被一起读取，而不用再另外进行读磁盘操作。另外，从相同数据块多次读取记录时，时间间隔越大，数据块存在于 InnoDB 或 XtraDB 缓冲池中的可能性就越小，所以应该尽量对相同数据块的记录进行连续读取，而“多范围读”优化技术就有这样的优点。

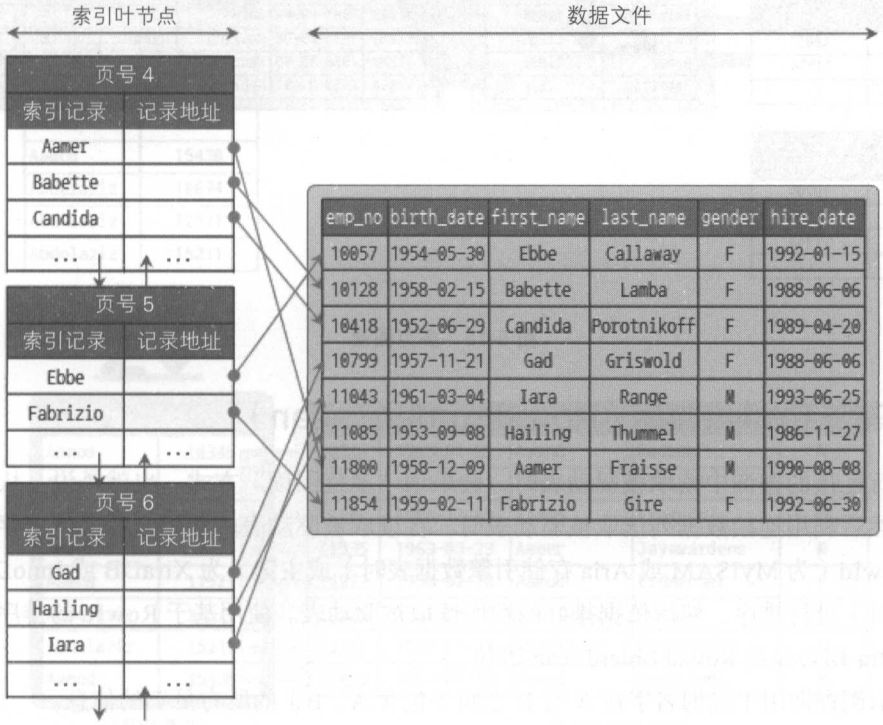


图 5-14 使用索引读取数据文件

图 5-15 描述了“多范围读”优化技术的工作原理。“多范围读”中，首先将通过索引读取的 employees 表的 first_name 列与 employees 表的主键（emp_no 列）值（地址）复制到 MRR 缓冲并进行排序，然后依据排好序的主键顺序读取 employees 数据表中的记录。图 5-14 中，由于要根据读索引的顺序读取数据文件中的记录，所以访问时与存储在实际数据文件中的记录顺序

毫无关系。而图 5-15 的“多范围读”中，对于通过索引读取的 first_name 与 emp_no 元组，要先按照 emp_no 列值顺序排序，然后再读取 employees 数据表，所以按照记录存储在实际数据文件中的顺序访问数据文件。

请注意，只有在 optimizer_switch 优化器选项中将 mrr 与 mrr_sort_keys 选项打开（ON），才可以使用“多范围读”优化技术。使用“多范围读”优化技术时，查询执行计划的 Extra 列会显示 Rowid-orderd scan、Key-orderd scan、Key-ordered Rowid-ordered scan 语句。也就是说，MariaDB 的“多范围读”优化采用 Rowid-orderd scan、Key-orderd scan、Key-ordered Rowid-ordered scan 这 3 种算法进行处理。

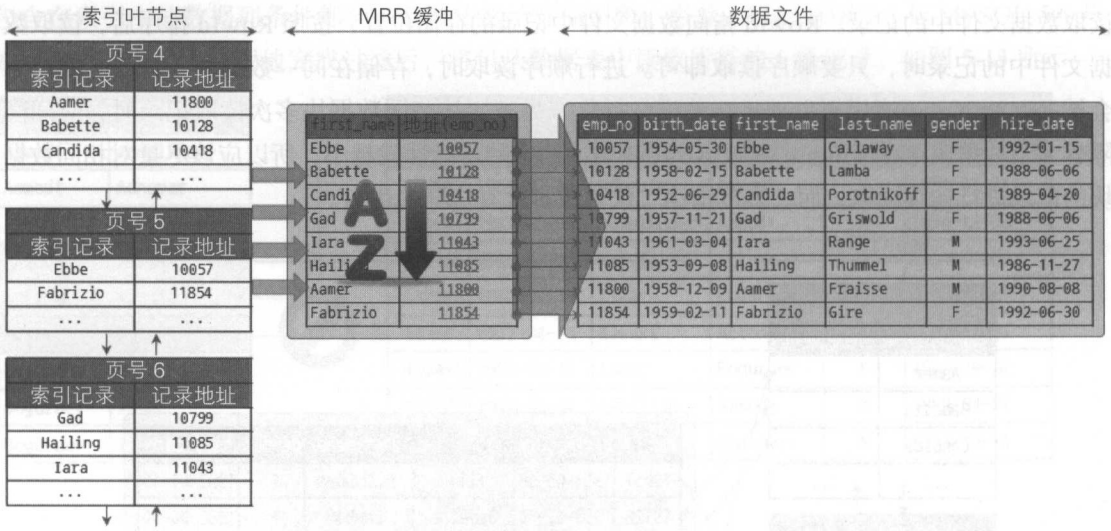


图 5-15 多范围读

5.7.1 基于 RowId 排序（Rowid-orderd scan）

在进行连接的查询中读取被驱动表时，会采用“多范围读”方式。也就是说，从驱动表读取记录，然后利用连接数据列读取被驱动表时，在读取被驱动表的索引之后，依据存储在索引尾部的 RowId（为 MyISAM 或 Aria 存储引擎数据表时）或主键（为 XtraDB 或 InnoDB 存储引擎数据表时）进行排序，然后依据排好的顺序读取被驱动表。使用基于 RowId 的排序时，执行计划的 Extra 列会显示 Rowid-orderd scan 语句。

下列示例查询用于查询名字在 A 与 B 之间（包含 A、B）的所有员工的信息。

```
SET optimizer_switch='mrr=ON';
SET optimizer_switch='mrr_sort_keys=ON';
EXPLAIN
SELECT * FROM employees WHERE first_name>='A' AND first_name<'B';
```

从示例查询的执行计划可以看到，使用 ix_firstname 索引查询 first_name 列值在 A 与 B 之间的所

有员工。此时由于 first_name 列范围很宽，优化器判断符合 WHERE 条件的记录数约为 43 000 条。

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	employees	range	ix_firstname	44	NULL	43230	Using index condition; Rowid-ordered scan

为了读取 employees 数据表的其余记录，要以随机 I/O 方式读取数据文件 43 000 次。为了优化该处理过程，先扫描 ix_firstname 索引，再依据 employees 数据表的 RowId 顺序对结果进行排序，然后执行多范围读操作。基于 RowId 进行排序时，查询执行计划的 Extra 列会显示 Rowid-ordered scan 语句。

图 5-16 表示执行示例查询时进行 Rowid-ordered scan 优化的过程。首先读取 ix_firstname 索引，检索符合 WHERE 条件的索引元组，然后在“多范围读”缓冲中，依据用作 RowId 的 emp_no 列（employees 数据表的主键）进行排序。并且按照 RowId 顺序访问实际 employees 数据表的数据文件，汇成最大批，使一次可以读取很多记录。

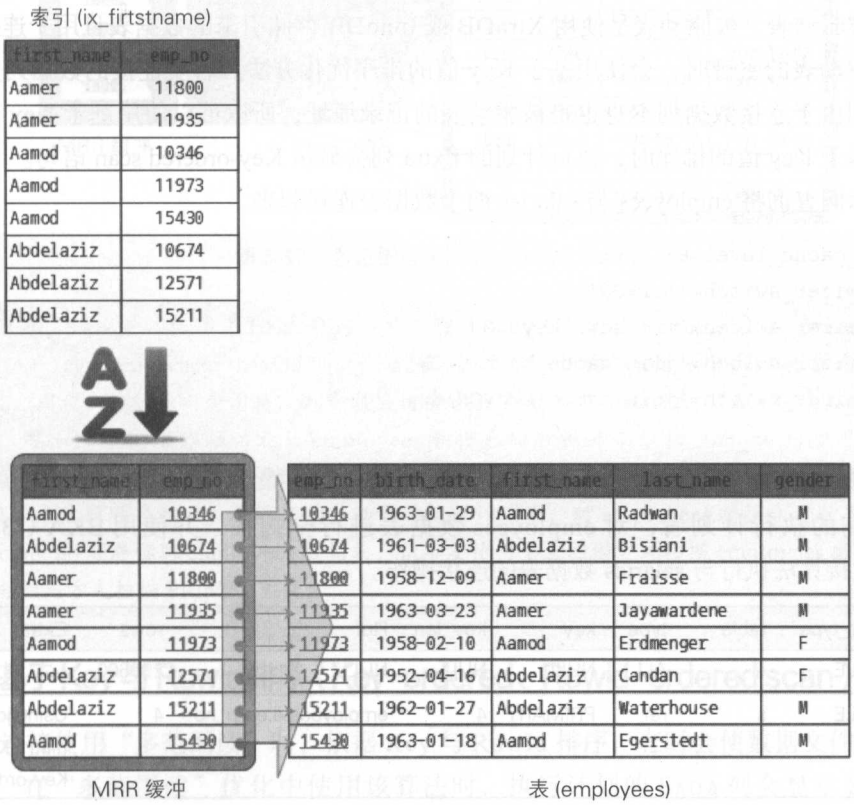


图 5-16 多范围读（Rowid-ordered scan）

接下来对示例查询稍加修改，将检索添加修改为 first_name='Matt'，如下所示。修改后的查询的执行计划是怎样的呢？

```
EXPLAIN SELECT * FROM employees WHERE first_name='Matt';
```

上述查询的执行计划如下所示。从执行计划看，预计要读取的全部记录数约为233条，并且不使用“多范围读”优化。也就是说，由于要读取的记录数不多，所以不使用“多范围读”，此时使用“多范围读”反而会降低查询性能。

id	select_type	table	Type	key	key_len	rows	Extra
1	SIMPLE	employees	Ref	ix_firstname	44	233	Using index condition

注意 MySQL 5.6也提供了相同的“多范围读”优化功能，但总是使用Rowid-ordered scan算法进行处理，不支持Key-ordered scan算法。因此，MySQL 5.6的执行计划中不会显示Rowid-ordered scan或Key-ordered scan语句，而只显示Using MRR语句。

5.7.2 基于Key排序（Key-ordered scan） MariaDB

与基于RowId排序不同，基于key排序时先读取驱动表，再对连接列进行排序，然后依据排序读取被驱动表。被驱动表是使用XtraDB或InnoDB存储引擎的数据表且用于连接条件的数据列是被驱动表的主键时，会使用基于Key值的排序优化方法。若要连接的数据列为被驱动表的主键，则由于连接数据列本身也是被驱动表的记录地址，所以可以使用基于Key值的排序优化。使用基于Key值的排序时，执行计划的Extra列会显示Key-ordered scan语句。

下列示例查询将employees与salaries两个数据表连接起来。

```
SET join_cache_level=8;
SET optimizer_switch='mrr=ON';
SET optimizer_switch='mrr_sort_keys=ON';
SET optimizer_switch='join_cache_hashed=ON';
SET optimizer_switch='join_cache_bka=ON';
EXPLAIN
SELECT * FROM employees e, salaries s WHERE s.emp_no=e.emp_no;
```

从查询的执行计划看，对employees数据表进行全扫描，并使用BKA（Batched Key Access）连接算法执行与salaries数据表的连接操作。

id	select_type	table	type	key	key_len	Ref	rows	Extra
1	SIMPLE	e	ALL	NULL	NULL	NULL	299600	
1	SIMPLE	s	ref	PRIMARY	4	employees.e.emp_no	4	Using join buffer (flat, BKAH join); Key-ordered scan

优化器从employees数据表读取emp_no数据列后，尝试读取salaries数据表。由于salaries数据表的主键为(emp_no+from_date)，所以优化器认为，先对employees数据表的emp_no列进行排序再读取salaries数据表会最大程度地减少随机I/O次数。因此，此处使用Key-ordered scan多范围读优化算法。一般而言，Key-ordered scan经常用于连接时的连接列是被驱动表的主键的情况。

图 5-17 表示使用 BKA 连接 employees 与 salaries 两个数据表的过程，以及使用 Key-ordered scan 多范围读优化算法的处理过程。

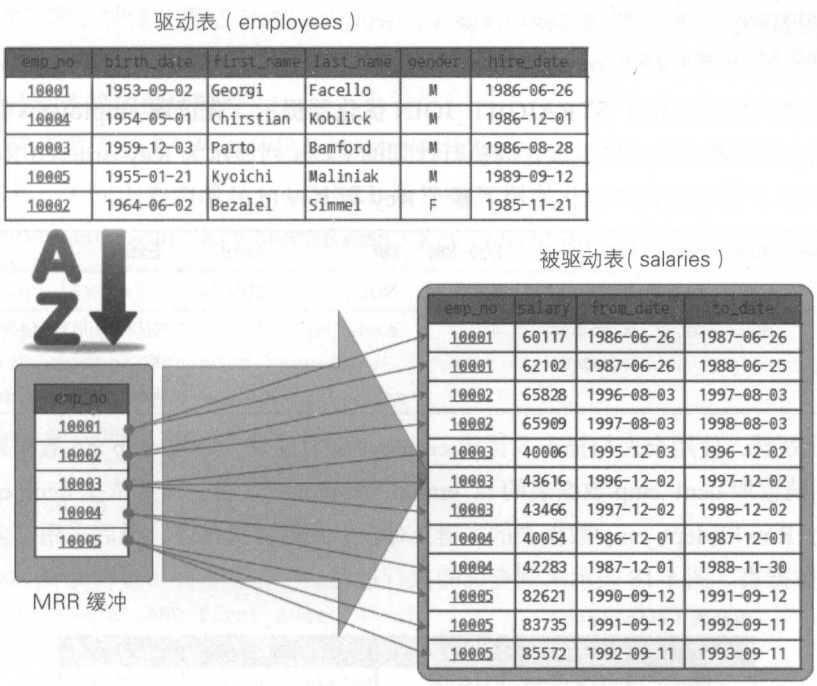


图 5-17 多范围读（Key-ordered scan）

注意 由上可知，示例查询使用了 BKA 散列连接，即执行示例查询时不仅要开启 optimizer_switch 的 mrr、mrr_sort_keys、join_cache_hashed、join_cache_bka 选项，还要将 join_cache_level 系统变量值设置为最大值 8。因此，执行示例查询前，先要调整 optimizer_switch 与 join_cache_level 的设置值。

此外，图 5-17 中，用作驱动表的 employees 数据表的数据并未依据 emp_no 列顺序进行排序。employees 数据表使用 XtraDB 或 InnoDB 存储引擎，在内部根据主键列 emp_no 排序。但图 5-17 中，emp_no 值并未依照顺序显示。原因在于，查询的执行计划中，对 employees 数据表进行了全表扫描。若要依照 emp_no 顺序读取 employees 数据表，请不要使用全表扫描方式扫描 employees 数据表，而要使用索引全扫描方式扫描 PRIMARY 键。

5.7.3 基于 Key 与 RowId 排序（Key-ordered、Rowid-ordered scan） MariaDB

BKA 连接使用“多范围读”时，依据 Key 与 RowId 排序，有时会使数据文件的随机读操作最小化。在“多范围读”优化中使用该算法时，执行计划的 Extra 列会显示 Key-ordered Rowid-ordered scan 语句。

下列示例查询只通过 emp_no 数据列连接 employees 与 dept_emp 两个数据表。

```
SET optimizer_switch='mrr=ON';
SET optimizer_switch='mrr_sort_keys=ON';
```

```
SET join_cache_level=8;
EXPLAIN
SELECT  STRAIGHT_JOIN * FROM employees e, dept_emp de
WHERE de.emp_no=e.emp_no;
```

上述示例查询特意使用了 STRAIGHT_JOIN 优化器提示，先读取 employees 数据表，然后读取 dept_emp 数据表进行连接。该查询执行计划的 Extra 列显示为 Key-ordered Rowid-ordered scan，由此可知，“多范围读”优化使用了基于 Key 与 RowId 的排序算法。

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	e	ALL	NULL	NULL	NULL	300024	
1	SIMPLE	de	ref	ix_empno_fromdate	4	e.emp_no	1	Using join buffer (flat, BKAH join); Key-ordered Rowid-ordered scan

从执行计划看，使用全表扫描方式读取 employees 数据表，查找 emp_no 数据列的值。先通过 emp_no 列值读取 dept_emp 数据表的 ix_empno_fromdate 索引，然后依据 dept_emp 数据表的 Key(emp_no)+RowId(dept_emp 数据表的主键) 对索引元组再次排序，最后采用“多范围读”读取 dept_emp 数据表。图 5-18 表示示例查询的执行过程。



图 5-18 多范围读（Key-ordered、Rowid-ordered scan）

5.7.4 “多范围读”优化与排序

“多范围读”的3种优化算法全部默认使用连接缓冲。后面讲解“块嵌套循环连接”（Block Nested Loop Join，BNL Join）时也会提到，使用连接缓冲执行连接时，从驱动表读取记录的顺序可能不会保持到连接的最后阶段。

下面向讲解“多范围读”的3种优化算法时使用的示例查询添加ORDER BY排序语句，并查看其执行计划。下列3个查询只有WHERE条件略有不同，基本结构与示例是一样的，它们分别使用Rowid-ordered scan、Key-ordered scan、Key-ordered Rowid-ordered scan算法。

```
SET join_cache_level=8;
SET optimizer_switch='mrr=ON';
SET optimizer_switch='mrr_sort_keys=ON';
SET optimizer_switch='join_cache_hashed=ON';
SET optimizer_switch='join_cache_bka=ON';

-- // Rowid-ordered scan
EXPLAIN
SELECT * FROM employees
WHERE first_name>='A' AND first_name<'B'
ORDER BY first_name;

-- // Key-ordered scan
EXPLAIN
SELECT * FROM employees e, salaries s
WHERE s.emp_no=e.emp_no AND e.emp_no BETWEEN 10001 AND 10020
ORDER BY e.emp_no

-- // Key-ordered Rowid-ordered scan
EXPLAIN
SELECT  STRAIGHT_JOIN * FROM employees e, dept_emp de
WHERE de.emp_no=e.emp_no AND e.emp_no BETWEEN 10001 AND 10100
ORDER BY e.emp_no;
```

上述示例的第一个查询使用Rowid-ordered scan算法，由于要使用first_name条件扫描ix_firstname索引，所以引入“多范围读”优化前，即使无“ORDER BY first_name”子句，也要用first_name排序并返回。但从引入“多范围读”优化开始，有无“ORDER BY first_name”子句会形成不同执行计划。下列是第一个示例查询的执行计划，该执行计划仅使用索引列添加排序，而不使用“多范围读”优化。

id	select_type	table	type	Key	key_len	rows	Extra
1	SIMPLE	employees	range	ix_firstname	44	43230	Using index condition

下列是第二个示例查询的执行计划，它使用 Key-ordered scan 算法。从执行计划可以看到，虽然使用了“多范围读”的 Key-ordered scan 算法，但由于存在 ORDER BY 子句，所以同时伴有临时表与另外的排序（Using filesort）处理。

id	select_type	table	type	key	key_len	rows	Extra
1	SIMPLE	e	range	PRIMARY	4	20	Using where; Using temporary; Using filesort
1	SIMPLE	s	ref	PRIMARY	4	9	Using join buffer (flat, BKAH join); Key-ordered scan

下列是第三个查询的执行计划，它也采用 Key-ordered Rowid-ordered scan 算法。但与第二个查询一样，先将连接结果存储到临时表，然后进行另外的排序处理。

id	select_type	table	type	key	key_len	rows	Extra
1	SIMPLE	e	range	PRIMARY	4	100	Using where; Using temporary; Using filesort
1	SIMPLE	de	ref	ix_empno_fromdate	4	1	Using join buffer (flat, BKAH join); Key-ordered Rowid-ordered scan

第二个与第三个查询都对 employees 数据表的主键（emp_no）进行了范围扫描。因此，我们推测连接结果是依据 emp_no 数据列排序的，但其实进行了“多范围读”优化处理，且在查询执行的最后阶段都对两个查询进行了另外的排序处理。第一个示例查询没有使用“多范围读”优化，依据索引顺序进行读取，且不进行另外的排序处理，这样花费的代价会更小。但第二个与第三个查询中，虽然要进行另外的排序处理，但使用“多范围读”优化能够获得更高效率。

注意 随着版本升级，MariaDB 服务器的优化器不断改善、进化，变得越来越复杂。即使对扫描索引的简单查询也无法像以前一样保证其最终查询结果依据索引顺序排序。尤其是查询中使用了“多范围读”优化、散列连接以及“块嵌套循环连接”等，必须明确指定所需的排序条件，这样才能得到希望的排序结果。

5.7.5 “多范围读”优化的注意事项

“多范围读”优化并非总能保证获得良好的性能。“多范围读”算法会对读入的索引元组先进行排序处理。数据量很少而可以全部放入 XtraDB 或 InnoDB 缓冲池时，“多范围读”算法的排序处理反而会占用更多系统资源，并影响系统性能。此外，带有 LIMIT n 或 ORDER BY LIMIT n 子句的查询中，若 n 取值较小，由于“多范围读”优化会读取许多不必要的索引元组并进行排序，这反而会降低处理性能。

为了进行“多范围读”，首先要对读入的索引元组进行排序，此时需要使用另外的内存空间。该内存空间大小由 mrr_buffer_size 系统变量设置，但若 MRR 缓冲尺寸太小，“多范围读”操作就会被划分为多个较小单位运行，因而性能很低；而 MRR 缓冲设置得太大时，又会发生内存浪费现象。并且，一个查询的执行过程中，若要连接多个数据表并执行多次“多范围读”优化，则要分配多个 MRR 缓冲。

“多范围读”优化在 BKA 连接算法中也会用到，后面讲解连接优化时会再次提到。此情形下，“多范围读”优化算法不使用 MRR 缓冲，而使用 BKA 连接算法管理的内存空间。在 BKA 连接中使用“多范围读”优化时，会使用连接缓冲（大小由 join_buffer_size 系统变量设置）。BKA 连接中多个数据表间的连接使用“多范围读”时，也要分配多次连接缓冲。像这样，分配多次连接缓冲时，要使用 join_cache_space_limit 系统变量限制连接缓冲的最大分配大小，以防浪费太多内存。

使用“多范围读”优化时，相关状态变量的值就会增加。通过检查下列状态变量的增减，可以监视“多范围读”的使用次数。

Handler_mrr_init

表示“多范围读”优化的使用次数。

Handler_mrr_key_refills

表示在“多范围读”优化中使用 Key-ordered scan 时，MRR 缓冲的重填次数。

Handler_mrr_rowid_refills

表示在“多范围读”优化中使用 Rowid-ordered scan 时，MRR 缓冲的重填次数。

“多范围读”相关内容可简单整理如下。

- “多范围读”优化可用于 range 访问方法或 BKA 连接。
- 只读取少量记录时，使用“多范围读”优化反而会降低处理性能。
- 执行“多范围读”优化时采用 Rowid-ordered scan、Key-ordered scan、Key-ordered Rowid-ordered scan 这 3 种算法之一。
- 只有开启（ON）optimizer_switch 优化器选项中的 mrr 与 mrr_sort_keys 选项，才能使用“多范围读”优化。

5.8 索引合并 MariaDB MySQL

通过索引读取数据时，每个数据表通常只能使用 1 个索引。但使用索引合并执行计划时，对于 1 个数据表可以使用 2 个以上的索引处理查询。查询中对于 1 个数据表的 WHERE 条件即使有多个，通常只用包含在 1 个索引中的数据列条件检索索引，而用其余条件对读取的记录进行检测。像这样，若只使用 1 个索引就能充分缩小处理范围，那么每个数据表只使用 1 个索引就非常高效了。但有多个 WHERE 条件且每个条件只能使用不同索引，而估计满足各条件的记

录数很多时，MariaDB 就选用索引合并执行计划。

下列查询中，WHERE 子句含有两个条件，employees 数据表的 first_name 列与 from_date 列都含有各自的索引（ix_firstname、ix_hiredate）。换言之，无论使用两个条件中的哪一个，都可以使用索引。

```
EXPLAIN
SELECT * FROM employees
WHERE first_name='Matt'
       AND hire_date BETWEEN '1995-01-01' AND '2000-01-01'
```

此时，MariaDB 优化器会先大致估计满足两个条件的记录数，然后使用记录数较少的条件。上述查询中，满足 first_name='Matt' 条件的记录（233 条）比满足 hire_date 列条件的记录数（34 028 条）更少。因此，MariaDB 优化器只检索 employees 数据表的 ix_firstname 索引。读取记录的其余所有列后，再检查是否满足 hire_date BETWEEN'1995-01-01'AND'2000-01-01' 条件，形成最终结果记录。

与预想的一样，上述查询执行时只使用 ix_firstname 索引。

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	employees	ref	ix_firstname	44	const	233	Using index condition; Using where

下面略微修改查询，从 employees 数据表查找 first_name 在 A 与 B 之间且在 1995-01-01 与 2000-01-01 之间入职的员工。执行该查询会怎样呢？

```
EXPLAIN
SELECT * FROM employees
WHERE first_name BETWEEN 'A' AND 'B'
     AND hire_date BETWEEN '1995-01-01' AND '2000-01-01'
```

如下所示，分别满足 first_name BETWEEN'A'AND'B' 条件与 hire_date BETWEEN'1995-01-01'AND'2000-01-01' 条件的记录数并不少。

```
SELECT COUNT(*) FROM employees WHERE first_name BETWEEN 'A' AND 'B' ;
==> 22039
SELECT COUNT(*) FROM employees WHERE hire_date BETWEEN '1995-01-01' AND '2000-01-01';
==> 34028
```

像这样，两个条件都不高效时，MariaDB 优化器会将两个条件分离，然后用各自的索引执行两次检索。之后合并两个结果集合，并将结果返回。从下列执行计划可以看到，上述查询使用了 index_merge 优化。

id	select_type	table	type	key	rows	Extra
1	SIMPLE	employees	index_merge	ix_firstname, ix_hiredate	9912	Using sort_intersect (ix_firstname,ix_hiredate); Using where

注意 示例查询的执行计划使用了 sort_intersection 优化算法，默认处于禁用状态。因此，为了查看执行计划，要设置 optimizer_switch 选项，启用 sort_intersection 算法。

```
SET optimizer_switch='index_merge_sort_intersection=on';
```

若使用了 index_merge，查询执行计划的 Extra 列总是显示以下 4 个语句之一。

- Using union
- Using intersect
- Using sort_union
- Using sort_intersection

5.8.1 Using union

WHERE 子句含有 2 个以上的条件，它们分别使用各自的索引。若由 OR 连接，则处理时会使用 Using union 算法。下列示例查询用于从 employees 数据表查询 first_name 列为 Matt 或 hire_date 列为 1987-03-31 的员工信息。

```
EXPLAIN
SELECT * FROM employees
WHERE first_name='Matt'
      OR hire_date='1987-03-31'
```

请注意，上述查询中 WHERE 的两个条件是由 OR 连接的。employees 数据表的 first_name 列与 hire_date 列分别建有 ix_firstname 与 ix_hiredate 索引。因此，first_name='Matt' 与 hire_date='1987-03-31' 条件可以使用各自的索引。该查询的执行计划使用 index_merge，如下所示。

id	select_type	table	type	key	rows	Extra
1	SIMPLE	employees	index_merge	ix_firstname, ix_hiredate	269	Using union (ix_firstname,ix_hiredate) ; Using where

查询执行计划的 Extra 列显示为 Using union(ix_firstname, ix_hiredate)，表示 index_merge 优化使用 union 算法将 ix_firstname 索引的检索结果与 ix_hiredate 索引的检索结果合并起来。此处的“合并”指获取两个集合的合集。

注意 其实，SQL 语句中 AND 运算符与 OR 运算符的差别相当大。两个条件由 AND 连接时，只要其中一个可以使用索引，就会使用索引范围扫描执行查询；而 WHERE 中的两个条件由 OR 连接时，只要其中一个条件无法正常使用索引，就只能使用全表扫描方式处理。

对上述示例查询稍加修改，如下所示，查看其执行计划可以很容易地确认以上事实。下列示例查询中，将 first_name 列的比较运算符由相等条件更改为无法使用索引的 first_name LIKE'%Matt' 条件。

```
EXPLAIN
SELECT * FROM employees
WHERE first_name LIKE '%Matt'
      OR hire_date='1987-03-31'
```

以上查询的执行计划如下，由执行计划可知，执行查询时使用全表扫描方式读取 employees 数据表。

id	select_type	table	type	Key	key_len	Ref	rows	Extra
1	SIMPLE	employees	ALL	NULL	NULL	NULL	299600	Using where

下面看看 index_merge 优化的 Using union 算法隐藏的秘密。示例查询中，若存在满足 first_name='Matt' 与 hire_date='1987-03-31' 条件的员工，则该员工信息既包含于 ix_firstname 索引的检索结果，也包含于 ix_hiredate 索引的检索结果。但查询结果不会对该员工信息显示两次，也不应该显示两次。MariaDB 服务器会将两个结果集合排序，并删除重复记录，但实际执行计划中不会显示排序信息。MariaDB 服务器为了删除重复记录，在内部进行了排序处理吗？

图 5-19 表示索引合并优化的 Using union 算法的工作过程。由图可知，工号为 13163 的员工既包含于 first_name 列的检索结果，也包含于 hire_date 列的检索结果。也就是说，它同时包含于两个结果集合，所以一定要删除其中一个。但 MariaDB 已经将 first_name 条件的检索结果与 hire_date 条件的检索结果依据主键分别进行了排序。如下所示，将示例查询分为两个独立的查询，分别查看它们的执行计划或查询结果。可以很容易地发现，索引的检索结果已经依据主键进行了排序。

```
SELECT * FROM employees WHERE first_name='Matt';
SELECT * FROM employees WHERE hire_date='1987-03-31';
```

因此，MariaDB 服务器会从两个集合将记录逐一取出，相互比较，并将 emp_no 主键列值重复的记录过滤。像这样，我们把这种将两个集合中的结果逐一取出并删除重复记录的“漏斗”称为优先级队列（Priority Queue）。这是 Using union 与 Using sort union 算法最大的不同。

5.8.2 Using sort_union

索引合并的 Using sort_union 算法与 Using union 的操作非常相似。下列示例查询与 Using union 的示例查询类似，只是 hire_date 列的比较条件由相等条件变为范围条件（BETWEEN）。

```
EXPLAIN
SELECT * FROM employees
WHERE first_name='Matt'
      OR hire_date BETWEEN '1987-03-01' AND '1987-03-31'
```

采用 Using union 说明中使用的方式，将上述示例查询分解为如下两个独立查询。

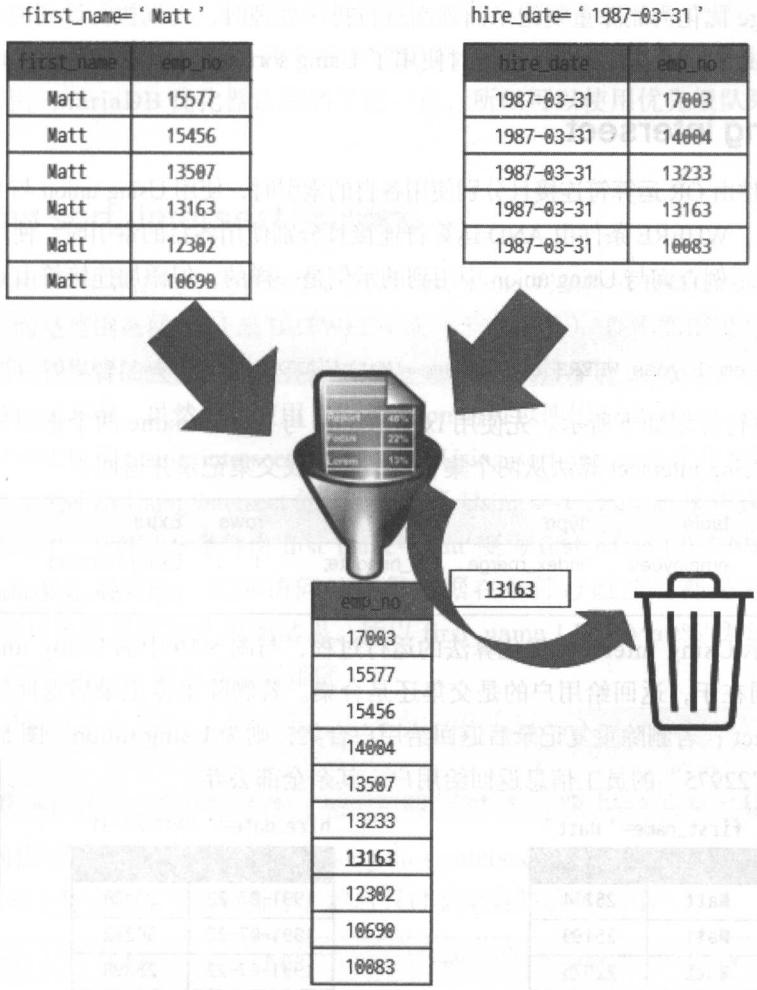


图 5-19 索引合并（Using union）

```
SELECT * FROM employees WHERE first_name='Matt';
SELECT * FROM employees WHERE hire_date BETWEEN '1987-03-01' AND '1987-03-31';
```

第一个查询的结果依据 emp_no 列进行排序，而第二个查询结果依据 hire_date 排序。也就是说，上述示例结果中，无法使用优先级队列删除重复记录。因此，为了从两个集合的结果中删除重复记录，MariaDB 服务器先依据 emp_no 列对各个集合排序，然后再删除重复记录。示例查询的执行计划如下所示。

Id	select_type	table	type	key	key_len	rows	Extra
1	SIMPLE	employees	index_merge	ix_firstname, ix_hiredate	44,3	3196	Using sort_union (ix_firstname,ix_hiredate); Using where

index_merge 优化为删除重复记录而强制进行排序处理时，查询执行计划的 Extra 列就会显示 Using sort_union 信息，表示执行查询时使用了 Using sort_union 算法。

5.8.3 Using intersect

WHERE 条件由 OR 运算符连接且分别使用各自的索引时，使用 Using union 与 Using sort_union 优化算法。反之，WHERE 条件由 AND 运算符连接且分别使用各自的索引时，使用 Using intersect 优化算法。下列示例查询与 Using union 中用到的示例是一样的，但添加连接符由 OR 变为 AND。

```
EXPLAIN
SELECT * FROM employees WHERE first_name='Matt' AND hire_date='1991-07-22';
```

该查询的执行计划如下所示。先使用 ix_hiredate 与 ix_firstname 两个索引分别创建结果集合，然后使用 Using intersect 算法从两个集合的结果查找交集记录并返回。

id	select_type	table	type	key	rows	Extra
1	SIMPLE	employees	index_merge	ix_hiredate, ix_firstname	1	Using intersect (ix_hiredate,ix_firstname); Using where

图 5-20 表示 Using intersect 优化算法的运行过程，与图 5-19 中的 Using union 算法几乎一样。两者的不同在于，返回给用户的是交集还是合集。若删除重复记录后返回给用户交集，则为 Using intersect；若删除重复记录后返回给用户合集，则为 Using union。图 5-20 中，只将两个集合的交集“22975”的员工信息返回给用户，其余全部丢弃。

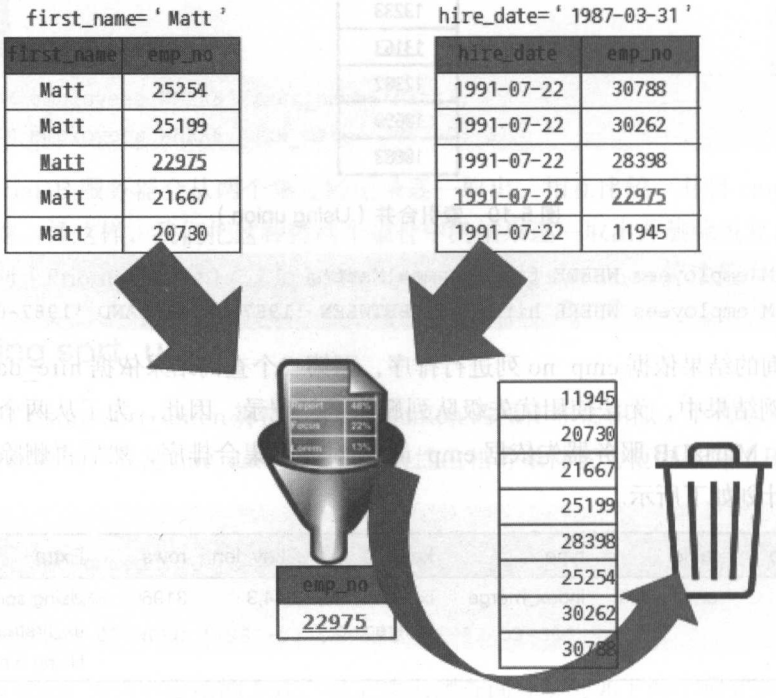


图 5-20 索引合并（Using intersect）

Using intersect 优化算法不需要为了丢弃多余记录与删除重复记录而进行另外的排序处理。与 Using union 中学过的一样，两个集合的结果已经依据 employees 数据表的主键 emp_no 列进行了排序。MariaDB 优化器认识到了这一点，所以可以使用优先级队列检索处理交集的重复记录。

5.8.4 Using sort_intersect MariaDB

Using intersect 优化算法使用的查询中，只要 first_name 列条件或 hire_date 列条件中有一个不是相等条件，而是范围条件（IN 或 BETWEEN 或大于、小于），就不能使用 Using intersect 优化算法。因为此时各集合的查询结果没有依据 emp_no 列进行排序。

MySQL 5.5 或 5.6 中，虽然无法使用 Using union 时可以使用 Using sort_union 先排序再合并，但并未针对无法使用 Using intersect 的情形提供 Using sort_intersect 优化算法。从 MariaDB 10.0 开始，针对无法使用 Using intersect 的情形提供了 Using sort_intersect 优化算法。

下列示例查询中，WHERE 条件由 first_name='Matt' 变为 first_name LIKE'Matt%'，执行该查询时就会产生范围处理操作。实际访问的结果数据在条件修改前后都是一样的，但由于 MariaDB 优化器只依据条件特性推测结果，所以 first_name LIKE'Matt%' 无法保证结果依据 employees 数据表的 emp_no 数据列进行排序。

```
SET optimizer_switch='index_merge_sort_intersection=on';
EXPLAIN
SELECT * FROM employees WHERE first_name LIKE 'Matt%' AND hire_date='1991-07-22';
```

上述查询的执行计划如下所示，使用 Using sort_intersect，即先使用 employees 数据表的主键 emp_no 列对两个集合的结果进行排序，然后只过滤交集并返回给用户。

id	select_type	table	type	key	rows	Extra
1	SIMPLE	employees	index_merge	ix_hiredate, ix_firstname	1	Using sort_intersect (ix_hiredate,ix_firstname); Using where

注意 示例中为了启用 Using sort_intersect 优化算法，开启（ON）index_merge_sort_intersection 优化器开关。我使用的 MariaDB 10.0.6 中，index_merge_sort_intersection 优化器开关默认值为 OFF，若不将其设置为 ON，则无法使用 Using sort_intersect 优化算法。请各位牢记，在自己所用的 MariaDB 中检查 sort_intersection 选项是否处于激活状态。

5.9 数据表连接 MariaDB MySQL

MySQL 5.5 与 MariaDB 5.2 之前的连接处理方式非常简单，仅有嵌套循环。但 MariaDB 5.3 与 MySQL 5.6 开始支持多种连接算法。下面对 MariaDB 中可用的连接类型与连接算法进行详细讲解。

5.9.1 连接类型

我们可以将连接类型大致分为 INNER JOIN 与 OUTER JOIN 两大类，OUTER JOIN 又可细分为 LEFT OUTER JOIN、RIGHT OUTER JOIN、FULL OUTER JOIN。此外，根据连接条件的显示方式，也可以将连接分为 NATURAL JOIN 和 CROSS JOIN（FULL JOIN、CARTESIAN JOIN）。

连接处理中，确定先读取哪个数据表是相当重要的，由此引起的处理工作量也大不一样。INNER JOIN 中，由于结果不随先读取的数据表的不同而不同，所以 MariaDB 优化器可以调整连接顺序并使用多种方法进行优化处理。但 OUTER JOIN 中，由于必须先读取用作 OUTER 的数据表，所以优化器无法选择连接顺序。

嵌套循环连接算法是 MariaDB 中使用的传统算法，下面介绍使用该算法进行各种连接的过程。

注意 半连接（Semi Join）与反连接（Anti Join）也属于连接。半连接无特定的语法形式，但查询中使用 IN(子查询) 或 EXISTS 时一般会用到，如下所示。

```
SELECT * FROM dept_emp
WHERE dept_no IN (SELECT dept_no FROM departments WHERE dept_name='Sales');

SELECT * FROM dept_emp de
WHERE EXISTS (SELECT 1 FROM departments d WHERE d.dept_no=de.dept_no);
```

半连接或反连接会在使用 [NOT] IN(子查询) 或 [NOT] EXISTS 的查询中用到，子查询优化中会进行详细讲解。

1. JOIN(INNER JOIN)

通常所说的“连接”指“内连接”（INNER JOIN），需要与“外连接”进行区别时，也会明确使用“内连接”（INNER JOIN）一词。MariaDB 传统上一直使用嵌套循环处理连接操作，类似于编写普通应用程序所使用的双层 FOR 或 WHILE 循环语句。

```
FOR ( record1 IN TABLE1 ){     // 外部循环（OUTER）
  FOR ( record2 IN TABLE2 ){   // 内部循环（INNER）
    IF ( record1.join_column == record2.join_column ){
      join_record_found(record1.*, record2.*);
    }else{
      join_record_notfound( );
    }
  }
}
```

由上述模拟代码可知，连接使用两层循环语句并根据条件连接两个数据表（模拟代码中的 FOR 循环语句并非指全表扫描）。两层 FOR 语句中，位于外侧 FOR 语句的数据表称为外部

（INNER）表，位于内侧 FOR 语句的数据表称为内部（INNER）表。并且，外部表要比内部表先读取，它在连接中起主导作用，称为驱动（Driving）表。内部表是连接中被“牵引”的表，因而也称为被驱动（Driven）表。

嵌套循环中，若最终选择的记录由内侧循环（INNER 表）决定，则称这种连接为内连接（INNER JOIN）。也就是说，运行双层嵌套循环时，只从 TABLE2（INNER 数据表）获取满足 IF(record1.join_column==record2.join_column) 条件的记录作为连接结果。

图 5-21 表示获取 INNER JOIN 执行结果的过程，其中 employees 为驱动表，salaries 为被驱动表。执行 INNER JOIN 的过程中，emp_no 为 100002 的记录在 salaries 数据表中并不存在，这种记录（仅存在于驱动表中的记录）不会包含到连接结果。

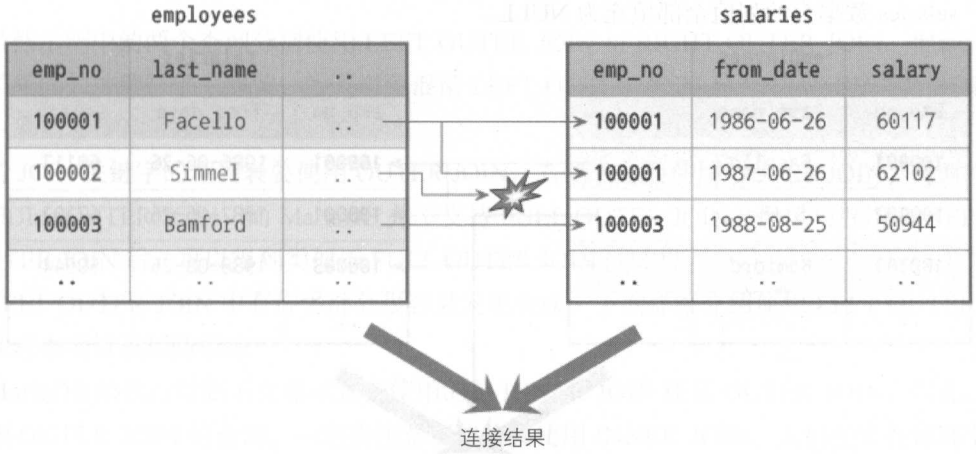


图 5-21 在 INNER JOIN 中获取结果的方法

2. OUTER JOIN

下面将 INNER JOIN 中用到的示意代码略作修改，如下所示。

```
FOR ( record1 IN TABLE1 ){      // 外部循环（OUTER）
  FOR ( record2 IN TABLE2 ){    // 内部循环（INNER）
    IF ( record1.join_column == record2.join_column ){
      join_record_found(record1.*, record2.*);
    }else{
      join_record_found(record1.*, NULL);
    }
  }
}
```

上述代码中，若 TABLE2 中有满足条件的记录，则如 INNER 连接一样创建结果，但 TABLE2（INNER 数据表）中不存在满足条件的记录时，获取的 TABLE2 列全部填充为 NULL。也就是说，INNER JOIN 中查找不到满足条件的记录时，会将 TABLE1 的结果全部丢弃；但 OUTER JOIN 中不会将 TABLE1 的结果丢弃，而将其如实包含到结果。

INNER 数据表对连接结果毫无影响，连接结果由 OUTER 数据表的内容决定，这是外连接的特征。虽然 OUTER 数据表与 INNER 数据表的关系（最具代表的是 1:M 的关系）会导致最终结果记录数增加，但 OUTER 数据表的记录不会因 INNER 数据表中不存在相应记录而被舍弃。

图 5-22 表示外连接的处理方式，emp_no 为 100002 的记录在 salaries 数据表中不存在，但它最终仍会包含到外连接的结果。当然，由于 salaries 数据表中不存在相应记录，所以最终结果中属于 salaries 数据表的列值全部填充为 NULL。

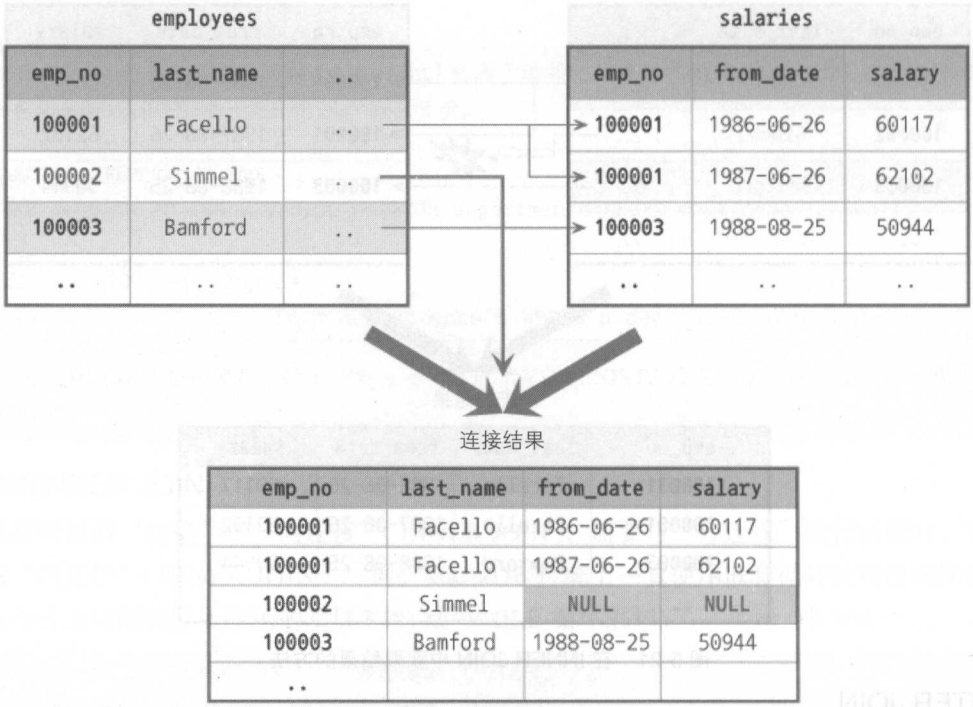


图 5-22 在 OUTER JOIN 中获取结果的方法

根据连接结果外部表位于连接的左侧还是右侧，OUTER JOIN 又可以细分为 LEFT OUTER JOIN、RIGHT OUTER JOIN、FULL OUTER JOIN。

```
SELECT *
FROM employees e
  LEFT OUTER JOIN salaries s ON s.emp_no = e.emp_no;

ELECT *
FROM salaries s
```

```
RIGHT OUTER JOIN employees e ON e.emp_no = s.emp_no;
```

上述示例中，第一个查询为 LEFT OUTER JOIN，第二个查询为 RIGHT OUTER JOIN，两个查询的不同之处如下所示。

- 第一个查询使用 LEFT OUTER JOIN，由于 employees 数据表在 LEFT OUTER JOIN 关键字的左侧，而 salaries 数据表在其右侧，所以 employees 为外部表。连接的最终结果由 employees 数据表的记录确定，与 salaries 数据表的记录存在与否无关。
- 第二个查询使用 RIGHT OUTER JOIN，由于 employees 数据表在 RIGHT OUTER JOIN 关键字的右侧，而 salaries 数据表在其左侧，所以 employees 为外部表。第二个查询的最终结果由 employees 数据表的记录确定，与 salaries 数据表的记录存在与否无关。

虽然示例中的两个查询分别使用 LEFT OUTER JOIN 与 RIGHT OUTER JOIN，但最终它们得到了相同的处理结果。因此，为了避免混淆 LEFT OUTER JOIN 与 RIGHT OUTER JOIN，一般统一采用 LEFT OUTER JOIN。

若 JOIN 关键字的左侧表要使用 OUTER JOIN，右侧表也要使用 OUTER JOIN，此时查询会使用 FULL OUTER JOIN，而 MariaDB 并不支持 FULL OUTER JOIN。但适当混用 INNER JOIN 与 OUTER JOIN 后，可以得到类似于 FULL OUTER JOIN 的效果。

LEFT OUTER JOIN 中有许多部分很容易发生失误。下面详细介绍使用 LEFT OUTER JOIN 时的注意事项以及预防措施。

MariaDB 的执行计划不会显示查询使用的是 INNER JOIN 还是 OUTER JOIN，因此，对于要使用 OUTER JOIN 的查询，一定要注意切勿使其使用 INNER JOIN。人们经常在该环节发生失误，OUTER JOIN 中，对于可能无记录一方数据表的条件，一定要在 LEFT JOIN 的 ON 子句中全部明确表示出来。否则，优化器可能会在内部将 OUTER JOIN 变为 INNER JOIN 进行处理。LEFT OUTER JOIN 的 ON 子句中明示的条件仅用于存在连接记录的情形。但 WHERE 子句中的条件会全部应用于连接结果，不受 OUTER JOIN 或 INNER JOIN 影响。查询中有使用 OUTER JOIN 连接的数据表时，若有可能，要习惯将所有条件明确放入 ON 子句。

```
SELECT *
FROM employees e
  LEFT OUTER JOIN salaries s ON s.emp_no=e.emp_no
WHERE s.salary > 5000;
```

上述查询的 LEFT OUTER JOIN 与 WHERE 子句相互冲突。由 OUTER JOIN 连接的数据表列的条件未在 ON 子句中明示，而在 WHERE 子句中明示，所以 MariaDB 服务器在优化查询时会先将其转换为如下查询，然后再执行。MariaDB 优化器转换查询后，返回的结果与原来查询的意图完全不同。

```
SELECT *
FROM employees e
```

```
  INNER JOIN salaries s ON s.emp_no=e.emp_no
WHERE s.salary > 5000;
```

对于上述这种形式的查询，只有修改为以下两种方式之一，才能使查询本身意图或结果得以明确。

```
--// 用简单的 OUTER JOIN 表示
SELECT *
FROM employees e
  LEFT OUTER JOIN salaries s ON s.emp_no=e.emp_no AND s.salary > 5000;

--// 用简单的 INNER JOIN 表示
SELECT *
FROM employees e
INNER JOIN salaries s ON s.emp_no=e.emp_no
WHERE s.salary > 5000;
```

不使用 LEFT OUTER JOIN 的查询中，即使不将检索条件或连接条件明确写在 WHERE 子句或 ON 子句中，也不会引发性能问题，或使结果有所不同。

注意 类似 Oracle 的 DBMS 中，数据列后可以使用“(+)”符号，表示有关 OUTER JOIN 数据表的条件。但 MariaDB 不允许使用这种标记法，仅支持 LEFT JOIN 或 LEFT OUTER JOIN 这样的 SQL 标准语法。

3. 笛卡儿连接

笛卡儿连接（cartesian join）又称 FULL JOIN 或 CROSS JOIN。一般而言，进行连接时都需要连接条件，用于在一个数据表中查找另一个数据表。但笛卡儿连接并不需要这样的连接条件，它获取两个数据表中的所有记录组合，并将其作为连接结果。笛卡儿连接中若要连接的数据表的记录数不多，只有 1~2 条，则不会有什么问题。但如果记录数增多，连接的结果条数就会呈几何级数增加，最终导致 MariaDB 服务器陷入无法响应的状态。

除了连接两端的数据表只有 1 条记录的查询，用于应用程序的笛卡儿连接大部分都会产生我们不想要的结果。要连接 N 个数据表的查询中，连接条件必须为 N-1 个（或大于 N-1 个），并且所有数据表必须在连接条件中使用 1 次以上，这样才能避免笛卡儿连接。请注意，要连接的数据表越多，连接条件越复杂，发生不想要的笛卡儿连接的可能性就越大。

```
SELECT * FROM departments WHERE dept_no='d001';
SELECT * FROM employees WHERE emp_no=1000001;
SELECT d.*, e.*

FROM departments d, employees e
WHERE dept_no = 'd001' AND emp_no=1000001;
```

此外，笛卡儿连接也用于使用 1 个查询统一执行只访问 1 条记录的多个查询（完全无关的

查询）。上述示例中，第一个与第二个查询各有 1 条访问记录，但它们完全无关。第三个查询将两个查询合并执行，使用 1 个查询访问两个数据表，一次就得到了想要的结果，但并无连接 employees 数据表与 departments 数据表的条件。像上面这样，使用笛卡儿连接可以快速执行两个查询，其效果就像执行 1 个查询一样。但若无法保证总有 1 条使用笛卡儿连接捆绑的两个单位查询所返回的记录，就有可能无法得到任何结果，也有可能得到比预期的更多的结果，希望各位注意。

SQL 标准中，CROSS JOIN 是与笛卡儿连接相同的连接方式；但 MariaDB 中，CROSS JOIN 是与 INNER JOIN 相同的连接方式。MariaDB 中，使用 CROSS JOIN 与 INNER JOIN 一样可以在 ON 或 WHERE 子句中加入连接条件，这样编写的 CROSS JOIN 以类似于 INNER JOIN 的方式工作。因此，MariaDB 中，CROSS JOIN 既可以是笛卡儿连接，也可以不是。下列两个示例将产生相同结果。

```
SELECT d.*, e.*
FROM departments d
  INNER JOIN employees e ON d.emp_no=e.emp_no;

SELECT d.*, e.*
FROM departments d
  CROSS JOIN employees e ON d.emp_no=e.emp_no;
```

其实，MariaDB 中，笛卡儿连接与内连接并不是通过语法进行区分的。若存在合适的连接条件，就会处理为内连接；若无连接条件，就成为笛卡儿连接。因此，使用 CROSS JOIN 或 INNER JOIN 时没有必要特意进行区分。

4. NATURAL JOIN

MariaDB 中有多种方法可以明确表示 INNER JOIN 条件。请看下列示例查询。

```
SELECT *
FROM employees e, salaries s
WHERE e.emp_no=s.emp_no;

SELECT *
FROM employees e
  INNER JOIN salaries s ON s.emp_no=e.emp_no;

SELECT *
FROM employees e
  INNER JOIN salaries s USING (emp_no);
```

上述 3 个示例查询是相同查询，只是标记方法略有不同。第三个查询中的 Using(emp_no) 与第二个查询中的 ON s.emp_no=e.emp_no 具有相同含义。只有两个要连接的数据表的连接列具有相同名称时，才可以使用 USING 关键字。

使用 NATURAL JOIN 也能得到与 INNER JOIN 一样的结果，但所用的表现方法略有不同。请看下列查询，了解并掌握 NATURAL JOIN 连接方法的特征。

```
SELECT *
FROM employees e
  NATURAL JOIN salaries s;
```

上述示例查询也用于连接 employees 数据表的 emp_no 列与 salaries 数据表的 emp_no 列。NATURAL JOIN 中，将存在于 employees 数据表与 salaries 数据表且名称相同的数据列全部用作连接条件。由于 employees 数据表与 salaries 数据表中名称相同的数据列只有 emp_no，所以 NATURAL JOIN salaries s 与 INNER JOIN salaries s ON s.emp_no=e.emp_no 具有相同含义。

虽然使用 NATURAL JOIN 时不用明确写出连接条件，使用起来比较方便，但查询会依据各数据表列名自动发生变化。也就是说，使用 NATURAL JOIN 连接的数据表使用相同列名时，会自动用作连接条件，必须时刻注意。另外，应用程序发生变化且数据表结构改变时，要查看是否有使用 NATURAL JOIN 连接的数据表，并比较数据表的列以查看有无同名数据列。这是一个相当繁琐的过程，极有可能产生较高的维护成本，只要知道有这么一种连接方式即可。

5.9.2 连接算法

MariaDB 5.3 之前的版本只支持使用简单嵌套循环算法的连接，但从 MariaDB 5.3 开始引入各种形态的连接算法。当然，MariaDB 5.3 或 MariaDB 5.5 中，这些新增连接算法的使用频率并不高，但相信随着 MariaDB 服务器版本的不断升级，这些连接算法的安全性正变得越来越高。

MariaDB 5.3 开始支持的连接算法有如下 5 种：

(1) 简单嵌套循环（Simple Nested Loop）
(2) 块嵌套循环（Block Nested Loop）
(3) 块嵌套循环散列（Block Nested Loop Hash）
(4) 块索引（Block Index Join，Batched Key Access）
(5) 块索引散列（Block Index Hash Join，Batched Key Access Hash）

下面逐个介绍这些连接算法的特征。虽然增加了不少连接算法，但这些算法大同小异，并不难理解。

1. 连接缓存级别（join_cache_level） MariaDB

MariaDB 10.0 提供了 4 种基于块的连接算法，每种算法又可以使用平坦（Flat）方式（这是一种传统方式，它将记录字段复制到连接缓冲）与增量（Incremental）方式（只将指针存储到连接缓冲）。因此，基于块的连接算法总共可分为如下 8 种：

(1) 块嵌套循环（Block Nested Loop-Flat）
(2) 块嵌套循环（Block Nested Loop-Incremental）
(3) 块嵌套循环（Block Nested Loop Hash-Flat）

(4) 块嵌套循环（Block Nested Loop Hash- Incremental）
(5) 批量主键访问（Batched Key Access-Flat）
(6) 批量主键访问（Batched Key Access- Incremental）
(7) 批量主键访问（Batched Key Access Hash-Flat）
(8) 批量主键访问（Batched Key Access Hash- Incremental）

3 个 optimizer_switch 选项（join_cache_incremental、join_cache_hashed、join_cache_bka）与 join_cache_level 系统变量将决定使用上述 8 种算法中的哪一种。若 optimizer_switch 中的 join_cache_bka 设置为 OFF，则禁用算法 5~8；若 join_cache_hashed 设置为 OFF，则禁用连接算法 3~4。算法 1 与 2 的块嵌套循环从以前版本开始就一直使用，它们是必需的，所以无法通过设置禁用。此外，若将 join_cache_incremental 设置为 OFF，则会使 2 与 4 以及 6 与 8 采用增量方式的连接算法进入禁用状态。默认 join_cache_bka、join_cache_hashed、join_cache_incremental 优化器开关全部初始化为 ON，即所有连接算法都处于可用状态。

不仅 optimizer_switch 选项会对优化器选择连接优化方法产生影响，join_cache_level 系统变量的设置也会对其产生影响。join_cache_level 系统变量的取值范围为 0~8，上述 1~8 连接算法中，只有算法编号小于 join_cache_level 变量值的连接算法才可用。也就是说，join_cache_level 设置为 1 时，只有 Nested Loop 基本连接算法以及 flat Block Nested Loop 连接算法可用。join_cache_level 设置为 8 时，上述所有连接算法都可用。

join_cache_level 系统变量的默认值为 1，如果不调整 join_cache_level 变量值，将只能使用块嵌套循环连接算法。MariaDB 10.0.6 的默认设置中，基于块的连接算法一般只用于 INNER JOIN 查询。若想在处理 OUTER JOIN 时也使用基于块的连接算法（使用连接缓冲），一定要将 outer_join_with_cache 优化器选项设置为 ON。若想使用基于块的连接算法处理 SEMI-JOIN，则一定要将 semijoin_with_cache 优化器选项设置为 ON。

由于 join_cache_level 与 optimizer_switch 系统变量既可以进行全局设置，也可以各连接为单位进行设置，所以可以根据用途灵活选择连接算法。

注意 MySQL 5.6.2 以前的版本中有一个 optimizer_join_cache_level 系统变量，它类似于 MariaDB 中的 join_cache_level 系统变量。MySQL 5.6 的 optimizer_join_cache_level 变量的取值范围为 0~4，optimizer_join_cache_level 设置值决定如何使用连接缓冲（平坦方式与增量方式），以及处理 OUTER JOIN 时是否使用块嵌套循环连接算法。

但从 MySQL 5.6.3 开始，optimizer_join_cache_level 系统变量不复存在，通过设置 optimizer_switch 系统变量决定使用块嵌套循环连接算法还是使用批量主键访问连接算法。从 MySQL 5.6.3 开始，通过设置 optimizer_switch 系统变量的 block_nested_loop、batched_key _access 项目为 ON 或 OFF，即可控制优化器选择的连接方式。

```
MySQL> SHOW GLOBAL VARIABLES LIKE 'optimizer_switch'\G
*************************** 1. row ***************************
Variable_name: optimizer_switch
        Value: ...,block_nested_loop=on,batched_key_access=off,...
```

```
--// 若想使用批量主键访问连接算法，请如下设置 optimizer_switch 变量
MySQL> SET GLOBAL optimizer_switch='batched_key_access=on';
```

2. 设置连接缓冲 MariaDB

如前所述，MariaDB 有多种连接算法。这些算法中，有的不需要使用另外的内存空间，比如简单的嵌套循环连接，而有的连接算法则需要使用另外的连接缓冲以提升连接性能。

多种连接算法中，基于块的连接（使用连接缓冲的连接）算法在内存中可用的最大连接缓冲大小由 join_buffer_size 系统变量确定。设置连接缓冲大小时，最小要使其可以存储 1 条连接目标记录。MariaDB 5.3 开始引入 join_cache_space_limit 系统变量，用于限制处理查询时最大可分配的连接缓冲的最大大小。

通过设置 join_cache_space_limit 系统变量限制连接缓冲，并将 optimizer_join_buffer_size 优化器的开关设置为 ON，能够以最优状态分配并使用连接缓冲。

```
SET optimizer_switch='optimize_join_buffer_size=ON';
SET join_cache_space_limit=5*1024*1024;
```

如上所示，打开 optimizer_join_buffer_size 优化器开关，并将 join_cache_space_limit 设置为 5 MB 后，根据查询执行计划预估的记录条数，在 join_cache_space_limit 设定的限制范围内为连接缓冲分配所需的内存空间。

3. 简单嵌套循环

MariaDB 5.1 或 MySQL 5.5 中，由于嵌套循环方式的种类并不多，所以一般不对其进行区分，而统称为“嵌套循环”。但从 MariaDB 5.3 开始，连接算法开始变得多样，需要使用单独术语称呼原有的嵌套循环方式的连接。于是在嵌套循环前添加了“Simple”（简单）一词，称之为“简单嵌套循环”（Simple Nested Loop，NL）。

简单嵌套循环连接是我们熟知的 MySQL 5.1 所用的连接算法。简单嵌套循环连接中，绝对不会使用连接缓冲（join_buffer_size）。简单嵌套循环是一个“古老”的算法，但使用合适的索引进行连接的查询中，仍会使用简单嵌套循环算法进行连接。此外，普通 Web 服务的 OLTP（On-line Transaction Processing，联机事务处理）查询中，也会适当使用索引进行连接，所以大部分查询会使用简单嵌套循环连接进行处理。

MariaDB 的嵌套循环连接也称为“Single-sweep multi join”，以前 MariaDB 的用户手册讲解连接方式时也使用“Single-sweep multi join”这一称呼，因其难以理解，故换用“嵌套循环连接”。“Single-sweep multi join”表示 FOR 或 WHILE 这样的循环嵌套次数与参与连接的数据表的个数一致。请看下列示例查询及其执行计划。

```
SELECT d.dept_name,e.first_name
FROM departments d, employees e, dept_emp de
WHERE de.dept_no=d.dept_no
```

```
AND e.emp_no=de.emp_no;
```

上述查询中连接 3 个数据表，其执行计划如下所示。

id	select_type	table	type	Key	key_len	Ref	rows	Extra
1	SIMPLE	d	index	ux_deptname	123	NULL	9	Using index
1	SIMPLE	de	ref	PRIMARY	12	employees.d.dept_no	18603	Using index
1	SIMPLE	e	eq_ref	PRIMARY	4	employees.de.emp_no	1	

从执行计划看，最先读取 d 数据表（department），然后依次读取 de 数据表（dept_emp）与 e 数据表（employees）。读取 de 数据表与 e 数据表时，ref 栏中显示哪些值用作比较条件。使用 FOR 循环表示该执行计划，如下所示。

```
FOR ( record1 IN departments ){
    FOR ( record2 IN dept_emp  &&  record2.dept_no = record1.dept_no ){
        FOR ( record3 IN employees && record3.emp_no = record2.emp_no ){
            RETURN {record1.dept_name, record3.first_name}
        }
    }
}
```

上述示意代码虽然有 3 层嵌套 FOR 循环，但从整体上看只有 1 个循环。我们将这种反复执行循环并以记录为单位依次读取所有连接目标表的方式称为“Single-sweep multi join”。MariaDB 连接结果先依据所读驱动表的顺序对记录进行排序再返回。连接中的被驱动表只依据读取驱动表记录的顺序进行检索（Lookup）。

4. 块嵌套循环

简单嵌套循环连接与块嵌套循环（Block Nested Loop，BNL）连接的最大不同在于是否使用连接缓冲（由 join_buffer_size 变量决定的连接缓冲），以及连接中以何种顺序连接驱动表与被驱动表。连接算法中使用“Block”一词表示连接中使用了另外的缓冲，而连接查询的执行计划中，Extra 列显示为“Using Join buffer”则表示该执行计划使用了连接缓冲。

连接时会根据驱动表中符合条件的记录数检索并处理被驱动表。也就是说，驱动表可以 1 次读完，而被驱动表则要读多次。比如，驱动表中符合条件的记录有 1000 条，若被驱动表的连接条件无法使用索引，则要对被驱动表进行 1000 次全表扫描，以查找要连接的记录。如果一个查询在检索被驱动表时无法使用索引，那么处理起来就会变得非常慢。MariaDB 优化器制定执行计划时会尽可能使检索被驱动表时可以使用索引。

若无论何种方式都无法避免对被驱动表进行全表扫描或索引全扫描，则优化器会先将从驱动表中读取的记录缓存到内存，然后将被驱动表与内存缓存进行连接处理。此时用到的内存缓存称为连接缓冲（Join buffer）。连接缓冲大小由 join_buffer_size 系统变量进行设置，连接完成后立即释放连接缓冲。

下列示例查询中有两个表要连接，两个表的连接条件位于 WHERE 子句，但连接条件中并不存在对两个表连接起“媒介”作用的条件。执行查询时会从 dept_emp 数据表读取所有满足 from_date>'2000-01-01' 条件的记录（10 616 条）、从 employees 数据表读取所有满足 emp_no<109004 的记录（99 003 条），并对其进行笛卡儿连接。

```
SELECT *
FROM dept_emp de, employees e
WHERE de.from_date>'2000-01-01' AND e.emp_no<109004;
```

图 5-23 表示不使用连接缓冲的情形下执行查询时获取结果的过程。假设连接时的 dept_emp 数据表为驱动表，employees 数据表为被驱动表。

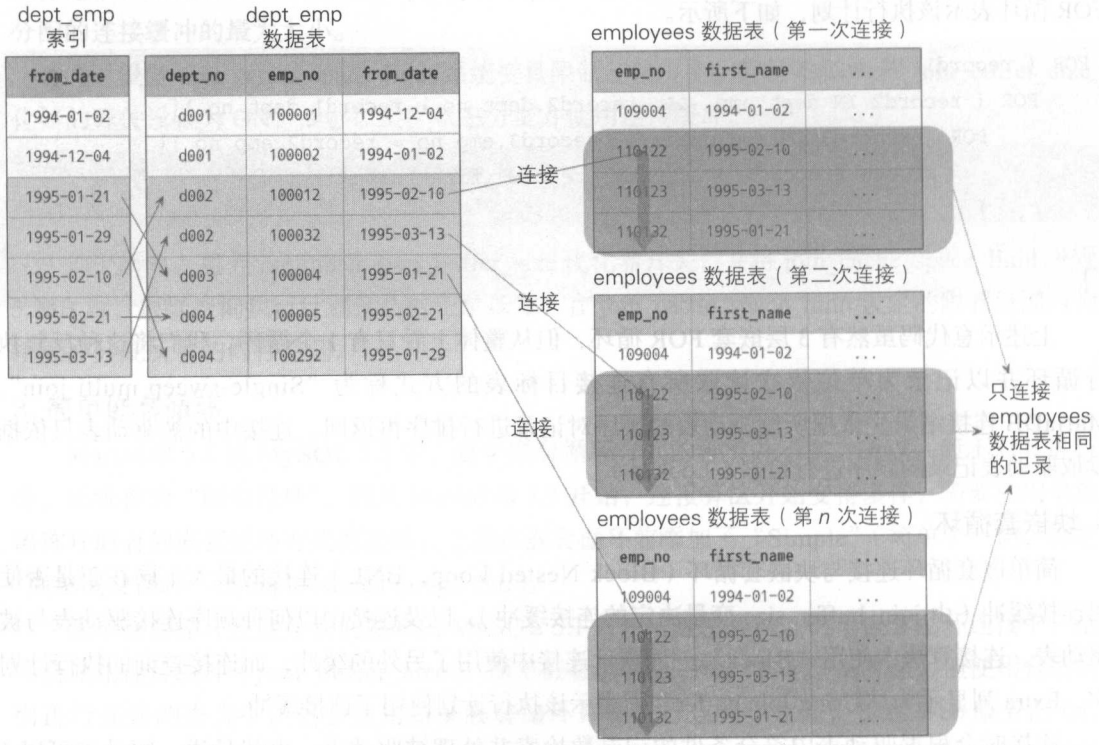


图 5-23 无连接缓冲条件时进行连接处理（Simple Nested Loop）

对于 dept_emp 数据表中满足 from_date>'2000-01-01' 条件的每条记录，都要从 employees 数据表获取满足 emp_no<109004 条件的 99 003 条记录。对于 dept_emp 数据表的每条记录，读取 employees 数据表时虽然每次从被驱动表获取的结果都一样，但仍然要将该操作执行 10 616 次。

若使用连接缓冲进行相同处理，情况又会如何呢？事实上，示例查询的执行计划如下所示。由执行计划可知，连接时，dept_emp 数据表充当驱动表。由 Extra 列显示的信息可知，读取 employees 数据表时使用了连接缓冲。

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	de	range	ix_fromdate	3		20550	Using where
1	SIMPLE	e	range	PRIMARY	4		148336	Using where; Using join buffer

图 5-24 表示查询的执行计划使用连接缓冲的过程，整个过程大致分为如下 4 个步骤。

(1) 使用 dept_emp 数据表的 ix_fromdate 索引检索满足 from_date>'2000-01-01' 条件的记录。

(2) 从 dept_emp 数据表读取连接需要的其余数据列，存入连接缓冲。

(3) 使用 employees 数据表的主键，检索满足 emp_no<109004 条件的记录。

(4) 将步骤 3 中检索的结果（employees）与步骤 2 中存入连接缓冲的记录（dept_emp）连接后，将结果返回给用户。

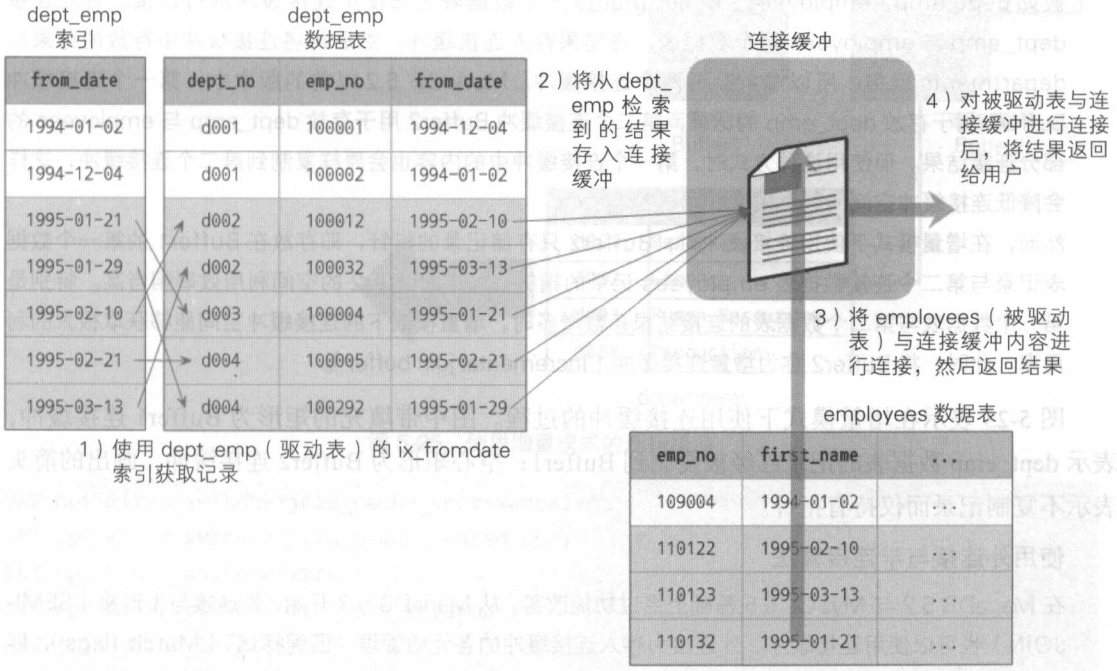


图 5-24 使用连接缓冲进行连接（Block Nested Loop）

图 5-24 中执行使用连接缓冲的查询时，连接顺序好像是颠倒的。上述说明中的步骤 4 以 employees 数据表的检索结果为基准，将其与 dept_emp 数据表的检索结果进行结合（合并）。从查询的执行计划看，dept_emp 数据表为驱动表，employees 数据表为被驱动表。实际的处理方式是先将驱动表的结果存储到连接缓冲，再读取被驱动表，然后从连接缓冲中查找满足条件的记录。执行连接后获取的结果通常由驱动表的顺序决定，但使用连接缓冲的连接中，结果的排序可能是无序的，请各位牢记。

注意 使用连接缓冲时，若首次读取的表结果太多而无法全部存入连接缓冲，则会反复执行上述 1~4 步。此外，由于连接缓冲只存储连接查询所需的数据列，而非存储记录中的所有数据列（查询执行时不必要的数据列），所以使用效率相当高。

为了更高效地使用连接缓冲，MariaDB 5.3 开始修改了存储在连接缓冲中的记录格式，新格式主要有以下优点。

处理可变长度数据列与 NULL 值时，进一步提高了空间的利用效率。

改善了对 NULL 值的处理，使之不再使用连接缓冲空间，并且在处理类似 VARCHAR 的可变长度数据列时，不必再使用 0 填充到最大长度。

支持连接缓冲的增量（Incremental）模式。

MariaDB 5.2 或 MySQL 5.5 之前版本中，只使用将查询所需元组（记录）全部存入连接缓冲的方式。但从 MariaDB 5.3 开始，连接 3 个以上的数据表时，会将重复内容从连接缓冲中删除，而只存储不重复的内容，这种方式称为增量（Incremental）模式。

假如 dept_emp、employees、departments 三个数据表全部使用连接缓冲进行连接。首先连接 dept_emp 与 employees 两个数据表，将结果存入连接缓冲。然后再将连接缓冲中存放的结果与 departments 连接，所以总共需要两个连接缓冲。MariaDB 5.2 以前的版本中，第一个连接缓冲 Buffer1 用于存放 dept_emp 的记录，第二个连接缓冲 Buffer2 用于存放 dept_emp 与 employees 的部分连接结果。但使用这种方式时，第一个连接缓冲中的内容也会原样复制到第二个连接缓冲，这样会降低连接缓冲空间的利用效率。

然而，在增量模式下使用连接缓冲时，Buffer2 只存储记录的指针，即存放在 Buffer1 的第一个数据表记录与第二个连接数据表 employees 记录的指针，所以 Buffer2 的空间利用效率相当高。特别是第一个数据表与第二个数据表的连接结果条数增多时，增量模式下的连接缓冲空间能够获取极大的利用率。此时，将 Buffer2 称为增量连接缓冲（Incremental join buffer）。

图 5-25 表示在增量模式下使用连接缓冲的过程。图中带填充的矩形为 Buffer1 连接缓冲，表示 dept_emp 数据表的记录已经被复制到 Buffer1；空心矩形为 Buffer2 连接缓冲，画出的箭头表示不复制记录而仅持有指针。

使用外连接与半连接算法

在 MariaDB 5.2 与 MySQL 5.5 基础上经过功能改善，从 MariaDB 5.3 开始，外连接与半连接（SEMI-JOIN）也可以使用连接缓冲。外连接为存入连接缓冲的各元组管理“匹配标记”（Match flags），匹配标记保存着是否在外连接数据表中发现了满足连接条件的记录。在外连接数据表中发现符合条件的记录时，匹配标记即被设为 TRUE。连接已完成而匹配标记为 FALSE 时，就会将外部表的所有列填充为 NULL 后返回。半连接中匹配标记的用法与此一致。

5. 块嵌套循环散列 MariaDB

MariaDB 5.3 首次引入了 MySQL 中的散列连接。虽然 MariaDB 5.3 与 10.0 中的散列连接所使用的散列连接算法都很简单，但以后会变得越来越好。MariaDB 的散列连接使用经典的散列连接算法，在用户手册中也称为 Block Nested Loop Hash（BNLH）。散列连接只用于相等比较条件的连接中，比如“tab1.fd1=tab2.fd2”。假设要连接 departments 数据表与 dept_emp 数据表。

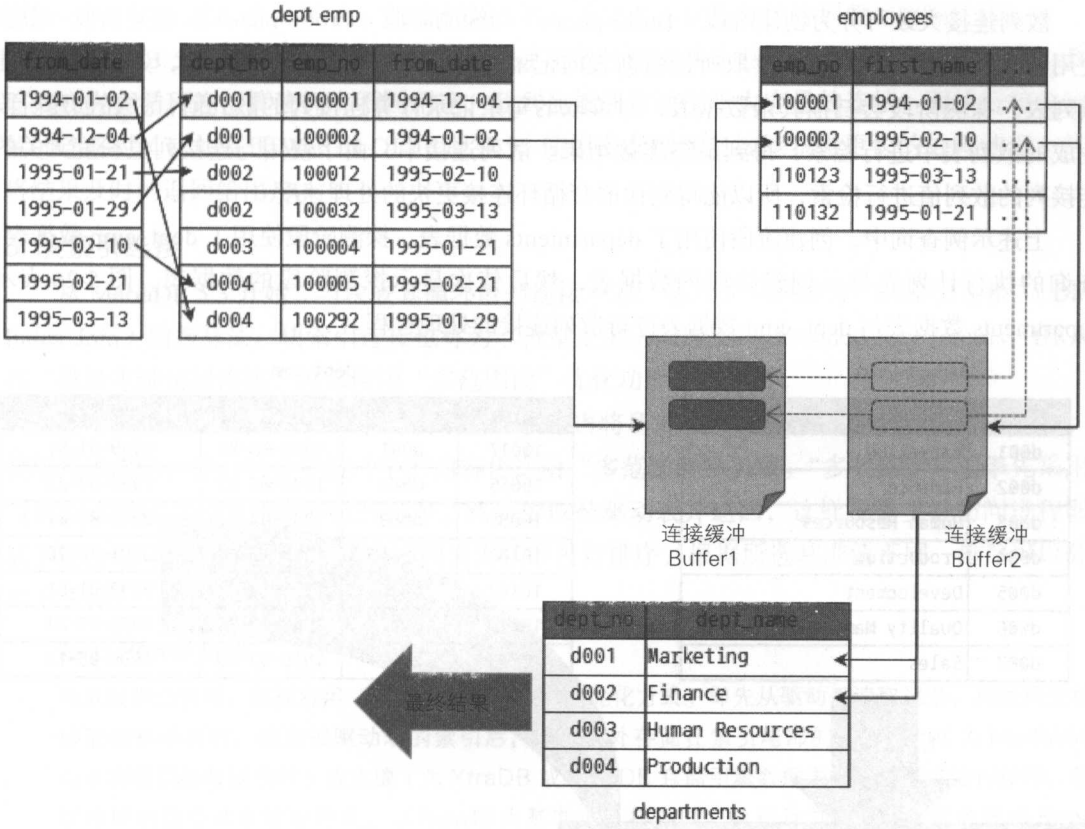


图 5-25 使用增量模式的连接缓冲

```
SET optimizer_switch='join_cache_incremental=on';
SET optimizer_switch='join_cache_hashed=on';
SET optimizer_switch='mrr=on';
SET optimizer_switch='mrr_sort_keys=on';
SET join_cache_level=8;

EXPLAIN
SELECT * FROM dept_emp de, departments d WHERE d.dept_no=de.dept_no;
```

上述查询通过 departments 数据表与 dept_emp 数据表的 dept_no 数据列的相等条件将两个数据表连接起来。该查询的执行计划如下所示，由执行计划可知，先读取 departments 数据表，然后再读取 dept_emp 数据表。

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	d	index	ux_deptname	122	NULL	9	Using index
1	SIMPLE	de	ref	PRIMARY	12	d.dept_no	20881	Using join buffer (flat, BKAH join); Key-ordered scan

散列连接大致可分为创建阶段（Build phase）与探测阶段（Probe phase）。创建阶段一般会使用记录数较少的数据表，读取所选数据表的记录，计算连接数据列的散列值，并使用其创建散列表。探测阶段会扫描其余数据表，对读取的每条记录计算连接列的散列值，并在创建阶段生成的散列表中进行检索，得到最终连接结果。散列连接中，由于使用与连接列值长度无关的连接列的散列值进行检索，所以能得到比嵌套循环连接更快的处理速度。

上述示例查询中，创建阶段使用了 departments 数据表，探测阶段使用了 dept_emp 数据表。查询的执行计划先显示创建阶段的数据表，然后依次显示探测阶段的数据表。图 5-26 表示 departments 数据表与 dept_emp 数据表进行散列连接的处理过程。

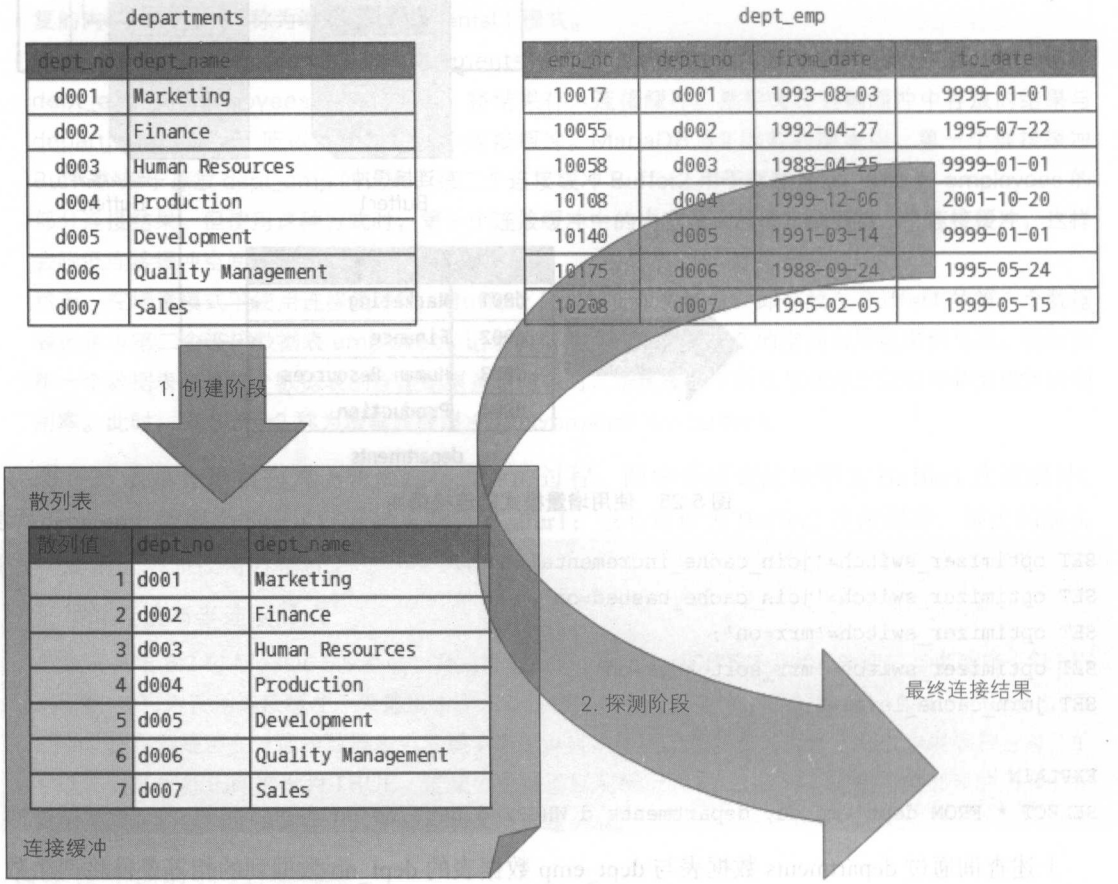


图 5-26 散列连接算法

散列连接的创建阶段生成的散列表存储到连接缓冲（join_buffer_size）。若创建阶段的数据表太大，生成的散列表比连接缓冲大，MariaDB 服务器就会将创建阶段与探测阶段切分为更多小的单位，并多次执行。因此，创建阶段使用的数据表的记录很多，可以将连接缓冲（join_buffer_size）适当设置得大一些，这有助于提高处理性能。此外，创建阶段的数据表很大时，由于会多次反复执行创建与探测阶段，所以 MariaDB 散列连接的名称为块嵌套循环散列。该名称

中的“块”表示使用连接缓冲，“嵌套循环”表示可能会多次反复执行创建与探测阶段。

MariaDB 5.3 或 10.0 支持散列连接，但使用散列连接并非总能得到较快的处理速度。散列连接或嵌套循环连接算法拥有不同用途，散列连接通常适用于大容量的连接，而执行较少记录量的连接时（比如基于 Web 的 OLTP 服务），使用嵌套循环连接能获得更快的处理速度，从而将结果及时的返回给用户。

6. 块索引连接

从 MariaDB 5.3 开始，与块嵌套循环散列连接一起，引入了一种称为“块索引连接”（Block Index Join）的新算法。块索引连接也称为“批量主键访问连接”（Batched Key Access，BKA），在“批量主键访问连接”内部使用“多范围读”优化功能进行连接。

在“批量主键访问连接”中先读取驱动表，再将所需数据列与连接列存储到连接缓冲。连接缓冲填满时，就会将连接缓冲中的内容传送给“多范围读”引擎。“多范围读”引擎先采用适当的方式对接收的缓冲内容进行排序，然后读取被驱动表并返回，这样批量主键访问进程即完成了连接处理。“多范围读”可以不同值为基准进行排序，根据所选基准值不同，排序可以分为如下 3 种：

基于 RowId 排序

读取被驱动表时，原样应用“多范围读”中介绍的优化方式。即先从驱动表读取记录，再使用连接列读取被驱动表时，读取被驱动表的索引后，以（额外存储在索引尾部的）RowId（为 MyISAM 或 Aria 存储引擎数据表时）或主键（为 XtraDB 或 InnoDB 存储引擎数据表时）为基准进行排序，然后以排好的顺序读取被驱动表。以 RowId 为基准进行排序时，执行计划的 Extra 列会显示 Rowid-ordered scan 信息。更多详细内容请参考 5.7.1 节与图 5-16。

基于 Key 排序 MariaDB

与“基于 RowId 排序”不同，基于 Key 进行的排序中，读取驱动表并使用连接列读取被驱动表的索引之前，先对连接列进行排序，然后依据排序读取被驱动表。被驱动表是使用 XtraDB 或 InnoDB 存储引擎的数据表，且连接条件中所用的数据列为被驱动表的主键时，会使用基于 Key 值的排序优化方法。若要连接的数据列为被驱动表的主键，由于连接列本身用作被驱动表记录的地址，所以可以使用这种优化。以 Key 值为基准进行排序时，执行计划的 Extra 列会显示 Key-ordered scan 信息。更多详细内容请参考 5.7.2 节与图 5-17。

基于 Key 与 RowId 排序 MariaDB

使用“多范围读”方式读取单一数据表时，其实不一定要对 Key 与 RowId 全部进行排序。但若通过连接读取被驱动表，则有时必须要对 Key 与 RowId 全部进行排序，并进行批量主键访问操作。更多详细内容请参考 5.7.3 节与图 5-18。

注意 “多范围读”引擎是指处理一个单位过程的代码块，它并不是类似于 XtraDB 与 InnoDB 存储引擎的存储引擎。我认为，将“多范围读”引擎视为函数（function）集合比较合适。

7. 块索引散列连接 MariaDB

块索引散列连接（Block Index Hash Join，Batched Key Access Hash）又称批量主键访问散列连接。顾名思义，它采用批量主键访问方式读取数据表，然后进行散列连接。该连接算法工作方式如下所示：

(1) 采用“多范围读”方式访问驱动表，生成散列表（创建阶段）；

(2) 读取被驱动表，检索散列表（探测阶段）并返回结果。

5.7 节和本节对“多范围读”以及散列连接相关内容进行了详细介绍，各位需要时可以参考。

5.9.3 连接的注意事项

MariaDB 的连接处理中，需要特别注意“执行结果的排序”和“INNER JOIN 与 OUTER JOIN 的选择”。

1. 连接执行结果的排序

一般而言，执行需要连接操作的查询时，从驱动表中读取记录的顺序就是整个查询结果的顺序。这也是嵌套循环连接方式的特征之一。请看下列示例查询。

```
SELECT de.dept_no, e.emp_no, e.first_name
FROM dept_emp de, employees e
WHERE e.emp_no=de.emp_no
  AND de.dept_no='d005' ;
```

从上述查询的执行计划看，处理该查询时，先使用 dept_emp 数据表的主键进行读取，然后使用从 dept_emp 数据表读取的结果检索 employees 数据表的主键。

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	de	ref	PRIMARY	12	const	53288	Using where; Using index
1	SIMPLE	e	eq_ref	PRIMARY	4	de.emp_no	1	

从执行计划可以进一步看出，由于 dept_emp 数据表的主键为 (dept_no+emp_no)，故可以预见 dept_emp 数据表的检索结果将先依据 dept_no 列序进行排序，然后再根据 emp_no 进行排序并返回。但由于该查询的 WHERE 条件含有 dept_no='d005' 这一固定条件，所以就像根据 emp_no 列进行排序一样。该查询虽然没有明确给出“ORDER BY de.emp_no ASC”条件，但可以得到依据 emp_no 列进行排序的效果。使用索引处理连接时，可以如上进行推测。

执行结果之所以按照以上顺序返回是因为，优化器从多种执行计划中选择了上述这种执行计划。如果优化器选择了其他执行计划，就可能得到不一样的结果。使用索引进行检索连接的查询中，若数据表的记录数很少或统计信息错误，优化器也可能选择其他执行计划。因此，应该尽量避免依赖于优化器所选执行计划的排序，因为查询的执行计划何时变更不得而知。数据表中的大部分记录某天会被删除，索引也会被删除或添加，从而引起执行计划发生变化，这些

情形都是很有可能发生的。

上述示例查询中，若希望依据员工号排序并返回结果，则一定要在查询语句中添加 ORDER BY de.emp_no ASC 子句，这样可以保证按指定顺序获取结果。即使查询中含有 ORDER BY 子句，优化器也并非总是进行排序处理。若执行计划可以保证排列顺序，则优化器就不会再进行另外的排序处理，而直接返回结果。如果无法保证排列顺序，那么就应该强制使用排序处理，以保证排列顺序。请注意，并不是只要使用 ORDER BY 子句，MariaDB 服务器就总是进行排序处理。

要在 SQL 查询中保证结果的排列顺序，只能使用 ORDER BY 子句，请各位牢记。

注意 Oracle 之类的 DBMS 提供了多种连接方法，选用不同连接方法所得结果的排列顺序也可能不同。因此，每次更新 Oracle DBMS 时，ORDER BY 总会发生问题。

MySQL 5.5 中，大部分连接采用嵌套循环连接算法进行处理，所以返回的查询结果都是依据驱动表进行排序的[①]。但从 MariaDB 5.3 与 MySQL 5.6 开始，连接缓冲的使用频率更高，而且引入了散列连接，所以连接结果的排列顺序不再与驱动表的顺序相关。总之，无论使用哪种 DBMS、何种连接算法，只有使用 ORDER BY 子句才能保证结果的排列顺序。

2. INNER JOIN 与 OUTER JOIN 的选择

INNER JOIN 中，只有连接两端的数据表中都存在的记录才会被返回；而 OUTER JOIN 中，只要记录存在于外部表就会被返回。从查询或数据表的结构看，对于可以不使用 OUTER JOIN 的情形，大部分时候都会使用 OUTER JOIN。好像有很多 DBMS 用户担心，使用 INNTER JOIN 后结果中不会有记录。OUTER JOIN 与 INNER JOIN 的用途不同，熟悉其用法后，根据要求的条件从中选用即可。

我们有时会认为采用 OUTER JOIN 处理查询速度会比较慢，所以编写查询时会强制使用 INNER JOIN。网上也经常有人问 OUTER JOIN 与 INNER JOIN 的性能差异问题。其实，若要获取的记录一样，OUTER JOIN 与 INNER JOIN 在查询性能上几乎没有差别。请比较下列两个查询，它们实际比较的条数或最终获取的结果条数一样（包含在查询中的 SQL_NO_CACHE 与 STRAIGHT_JOIN 提示会使条件一样。后文会详细讲解查询提示，此处先省略）。

```
SELECT SQL_NO_CACHE STRAIGHT_JOIN COUNT(*)
FROM dept_emp de
  INNER JOIN employees e ON e.emp_no=de.emp_no;

SELECT SQL_NO_CACHE STRAIGHT_JOIN COUNT(*)
FROM dept_emp de
  LEFT JOIN employees e ON e.emp_no=de.emp_no;
```

从我 PC 中的测试结果看，INNER JOIN 执行花费的平均时间约为 0.37 秒，OUTER JOIN

① 当然，MySQL 5.5 中使用连接缓冲进行嵌套循环连接时，结果记录的排列顺序可能与驱动表的读取顺序不一致。但 MySQL 5.5 中，由于连接缓冲只用于不使用索引的连接，所以连接缓冲并不常用。

约为 0.38 秒。OUTER JOIN 中，由于需要逐一执行另外的触发器条件，以判断进行连接的第二个数据表（employees）中是否存在相应记录，所以多花费了 0.01 秒。除此之外，两者之间不存在任何明显的性能差异。

希望各位选择 INNER JOIN 与 OUTER JOIN 时，不要从性能方面进行考量，而要根据业务条件进行选择。如果担心记录不会包含在结果中，则要从分析数据表的结构与数据的特征入手，选择使用 INNER JOIN 还是 OUTER JOIN。没有准确掌握数据结构与特征的情形下，使用 OUTER JOIN 可能会在画面上显示错误结果。

5.10 子查询 MariaDB MySQL

MySQL 5.5 以前的版本中，子查询的优化有很多问题。与 Oracle 这类商用 RDBMS 相比，MySQL 服务器的子查询优化相当差。对于以半连接形式使用的子查询（比如 IN(subquery)），MySQL 服务器只有一种优化方法，即将其转换为 EXISTS 再执行。无论是相关子查询（Correlated subquery）还是无关子查询（Non-correlated subquery），全都使用该方法进行处理。半连接中完全没有可以先处理子查询的方法。

从 MariaDB 5.3 开始，子查询的优化得到大幅改善。特别是在 [NOT] IN(subquery) 形式的半连接中使用的子查询，其优化功能得到大大增强。子查询能够以多种形式用于 SQL 语句，下面介绍如何区分各种子查询的类型以及如何进行优化。

5.10.1 半连接子查询优化

首先讲解 MySQL 服务器中“臭名昭著”的半连接子查询。事实上，可能许多读者喜欢使用这种形式的查询。

```
SELECT ...
FROM outer_tables
WHERE expr IN (SELECT ... FROM inner_tables ...)
```

一提到连接，我们一般只想到在 FROM 子句中罗列多个数据表或使用 JOIN 关键字的情形。但像上述示例一样只带有半边连接性质的查询也包含于连接，我们称之为“半连接”。半连接既可以使用常见的 JOIN 语法进行标记，也可以使用示例中的形式（IN(subquery)）。其实，半连接没有特别的语法特征，所以难以区分。

对于 IN(subquery) 形式的查询，MySQL 5.5 之前版本只采用如下处理形式。但如下所示，采用 EXISTS 形式进行处理有时效率比较高，而更多时候处理效率并不高。因此，MySQL 5.5 之前版本会有意将这种查询转换为连接等进行手工调优。

```
SELECT ...
FROM outer_tables
```

```
WHERE EXISTS (SELECT 1 FROM inner_tables WHERE inner_tables.expr=outer_tables.expr)

SELECT ...
FROM outer_tables o, inner_table i
WHERE o.expr=i.expr
```

从 MariaDB 5.3 开始，优化器会自动进行这样的优化处理。当然，优化器的性能也得到了很大改善，提供了比手工调优更多样的优化方法。MariaDB 10.0 对于半连接形式的子查询有下列 5 种优化方法。为了避免混淆，讲解时各优化名采用用户手册中原有的英文名称。

- Table pullout 优化
- FirstMatch 优化
- Semi-join Materialization 优化
- LooseScan 优化
- DuplicateWeedout 优化

为了使用上述 5 种优化方法对子查询进行优化（重写查询，Query Rewrite），要遵守下列几项限制条件。也就是说，只有满足下列条件的子查询，才可以应用后面介绍的半连接优化方法。

- 查询中使用 IN(subquery) 或 =ANY(subquery) 形式的条件时
- 子查询为不包含 UNION 的单一 SELECT 时
- 子查询不带有聚合函数（SUM 或 COUNT 等）与 HAVING 子句时
- 子查询的 WHERE 条件与外部查询的其他条件通过 AND 运算符进行连接时
- 子查询不是使用连接的 UPDATE 或 DELETE 语句时
- 不使用事先制定的执行计划时（使用 PreparedStatement 时重新使用执行计划）
- 外部查询与子查询都使用实际存在的数据表时（对于“SELECT 1”或“SELECT 1 FROM dual”这种不使用数据表的或者使用虚拟表的查询，不能使用半连接优化）
- 外部查询与子查询都不使用 STRAIGHT_JOIN 提示时

此外，从 MariaDB 源码看，还定义了几个条件，但是对于 SQL 级别的用户，只要记住上面这些条件即可。MariaDB 10.0 的自动优化中，一部分是可以通过手动方式修改查询的，而对于复杂形式的子查询优化，MariaDB 优化器也可以自动进行处理，目前已变得相当完善。但世间没有十全十美的事物，优化器也不是完美无瑕的，我们不能 100% 地相信它。为了更好地使用日益复杂化的优化器，一定要学习有关优化器及优化算法的内容，与此同时，需要注意的事项也越来越多。下面逐条学习半连接子查询的优化策略。

1. Table pullout 优化

Table pullout 优化中，首先将用于半连接子查询的数据表抽取到外部查询，然后将查询改写为连接查询。引入子查询优化之前，Table pullout 优化是最具代表性的手动查询调优方法。

下列示例查询用于查询部门号为 d009 的所有员工。在查询中使用 IN(subquery) 形式的半连接是最常见的做法。

```
EXPLAIN
SELECT * FROM employees e
WHERE e.emp_no IN (SELECT de.emp_no FROM dept_emp de WHERE de.dept_no='d009');
```

MariaDB 5.3 或 MySQL 5.6 以前版本中，上述查询的执行计划如下所示（下列执行计划是 MySQL 5.5 提供的）。

id	select_type	table	type	Key	Ref	rows	Extra
1	PRIMARY	e	ALL	NULL	NULL	300252	Using where
2	DEPENDENT SUBQUERY	de	eq_ref	PRIMARY	const,func	1	Using where; Using index

MySQL 5.5 中制定的执行计划确实很糟糕。首先对 employees 数据表进行全扫描，获取 emp_no 列值，然后逐个检查 emp_no 是否位于子查询的结果。由于从 employees 数据表总共读取了 300 252 条记录，所以进行条件检测时也要执行 300 252 次子查询。请注意，上述执行计划中，两行 id 字段值是不一样的。

上述示例查询在 MariaDB 10.0 中的执行计划如下所示。

id	select_type	table	type	key	ref	rows	Extra
1	PRIMARY	de	ref	PRIMARY	const	43004	Using where; Using index
1	PRIMARY	e	eq_ref	PRIMARY	de.emp_no	1	

该执行计划与前面执行计划最大的不同是，表格中两行 id 字段值是一样的。这意味着两个查询不是以子查询形式，而是以连接方式执行的。从查询的执行计划看，执行时先使用索引范围扫描从 dept_emp 数据表读取 43 004 条记录，然后以相同次数查询 employees 数据表。仅从执行查询时读取的记录数看，处理量已减少到 10%。

使用 Table pullout 优化时，执行计划的 Extra 列不会显示类似于 Using table pullout 的语句。判断是否使用了 Table pullout 优化的最简单的方法是，比较执行计划中相应数据表的 id 字段值是否相同（与此同时，Extra 字段未显示任何信息）。判断是否使用了 Table pullout 优化的更准确的方法是，使用 EXPLAIN EXTENDED 命令。

```
EXPLAIN EXTENDED
SELECT * FROM employees e
WHERE e.emp_no IN (SELECT de.emp_no FROM dept_emp de WHERE de.dept_no='d009');
=> 2 rows in set, 1 warning (0.00 sec)

SHOW WARNINGS;
```

使用 EXPLAIN EXTENDED 命令查看执行计划后，再使用 SHOW WARNINGS 命令查看 Warning 信息，显示内容如下。为了便于阅读，我对显示的内容进行了编辑。实际信息显示于 1 行，很难读懂。

```
SELECT
  'employees'.'e'.'emp_no' AS 'emp_no',
  'employees'.'e'.'birth_date' AS 'birth_date',
  'employees'.'e'.'first_name' AS 'first_name',
  'employees'.'e'.'last_name' AS 'last_name',
  'employees'.'e'.'gender' AS 'gender',
  'employees'.'e'.'hire_date' AS 'hire_date'
FROM 'employees'.'dept_emp' 'de'
  JOIN 'employees'.'employees' 'e'
WHERE (('employees'.'e'.'emp_no' = 'employees'.'de'.'emp_no')
  AND ('employees'.'de'.'dept_no' = 'd009'))
```

从上述 EXPLAIN EXTENDED+SHOW WARNINGS 命令的执行结果看，虽然显示的是 SQL 语句，但与刚才访问执行计划时输入的 SQL 语句是不同的。也就是说，使用 EXPLAIN EXTENDED 命令显示的查询语句是优化器对原先输入的 SQL 语句进行优化改写后的结果。通过该结果可以判断半连接子查询语句先经过连接优化处理，然后再执行。

注意 使用 SHOW WARNINGS 命令显示的结果与实际的 SQL 语句非常接近。上述示例显示的结果与 SQL 语法几乎完全一样，但使用 EXPLAIN EXTENDED 命令显示的内容其实是经过改造后的查询，即优化器对输入的 SQL 语句进行分析后，将分析结果以可读形式呈现。使用 EXPLAIN EXTENDED 命令显示的内容中，也可能包含优化器的优化内容（以“<...>”形式）。

Table pullout 优化并非在所有形式的子查询中都可以使用。下列是使用 Table pullout 优化的几种限制条件及其主要特征。

- Table pullout 优化只用于半连接子查询。
- 子查询使用 UNIQUE 索引或主键检索结果只有 1 条时，才可使用 Table pullout 优化。
- Table pullout 优化可用且查询中也有其他优化方法可以使用时，MariaDB 会最大限度地使用 Table pullout 优化。
- 进行 Table pullout 优化时，先将子查询的数据表置于外查询，然后进行连接操作。若子查询中的所有数据表都可以抽取到外查询，子查询本身将不复存在。
- MySQL 调优经常建议“尽量使用连接处理子查询”，Table pullout 优化正好遵循了这一点。有了 Table pullout 优化，使用子查询时就不必再分解为连接进行处理。

还有一点需要注意，与其他优化算法不同，Table pullout 优化无独立的优化器开关。也就是说，我们无法通过关闭相应的优化器开关以禁用 Table pullout。若必须关闭 Table pullout 优化，只要将 semijoin 优化器开关设置为 OFF 即可。此时不仅关闭 Table pullout 优化，后面将要讲解的半连接优化也会一起关闭，请各位牢记。

```
SET optimizer_switch='semijoin=OFF';
```

2. FirstMatch 优化

FirstMatch 也是用于半连接子查询优化的方法，执行时，将 IN(subquery) 形式的半连接优化为 EXISTS(subquery) 形式。下列示例查询从名字为 Matt 的员工中选出 1995 年 1 月 1 日 ~30 日之间有职称变动的员工。

```
EXPLAIN SELECT *
FROM employees e WHERE e.first_name='Matt'
  AND e.emp_no IN (
    SELECT t.emp_no FROM titles t
    WHERE t.from_date BETWEEN '1995-01-01' AND '1995-01-30'
  );
```

上述示例查询的执行计划如下所示。执行计划中需要注意的是，id 字段值全为 1，并且 Extra 列显示 FirstMatch(e) 语句。

id	select_type	table	type	key	key_len	ref	rows	Extra
1	PRIMARY	e	ref	ix_firstname	44	const	233	Using index condition
1	PRIMARY	t	ref	PRIMARY	4	e.emp_no	1	Using where; Using index; FirstMatch(e)

由执行计划的 id 字段值全为 1 可知，上述 FirstMatch 优化示例中，titles 数据表并未以子查询形式执行，而是处理为连接。FirstMatch(e) 语句表示，对于 employees 数据表的记录，只要在 titles 数据表中找到 1 条满足条件的记录，就不再继续检索 titles 数据表。从语义上看，这与 EXISTS(subquery) 的处理方式是一样的。但 FirstMatch 以连接而非子查询的方式执行，并且执行优化时只检索满足条件的第一个记录。图 5-27 表示上述执行计划的处理方式。

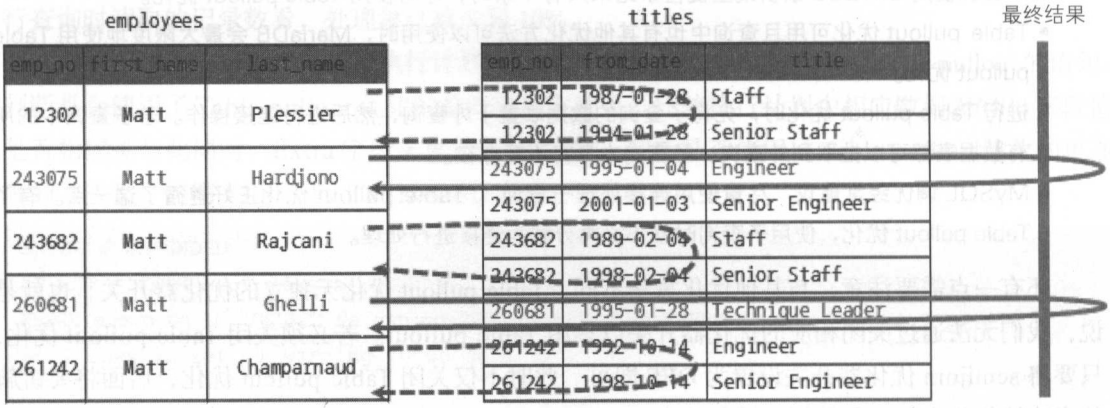


图 5-27 FirstMatch 优化的工作方式

首先使用 ix_firstname 索引，采用范围扫描方式从 employees 数据表读取 first_name 列值为 Matt 的员工信息，读取结果为图 5-27 左侧的 employees 数据表。然后将 first_name 为 Matt 且员

工号为 12302 的记录与 titles 数据表进行连接，从 titles 数据表的 from_date 列查找满足“t.from_date BETWEEN'1995-01-01'AND'1995-01-30'”条件的记录。由于 12302 号员工的 from_date 不满足上述条件，所以它不会被返回给用户。接下来，读取工号为 243075 的员工记录，与 titles 数据表进行连接，从连接的 titles 记录中检查 from_date 是否满足给定条件。此时找到满足条件的第一条记录，所以不再继续检索 titles 数据表，而将员工号为 243075 的记录作为最终结果并返回。

如图 5-27 所示，FirstMatch 优化执行的处理逻辑与 MySQL 5.5 优化方法中的 IN-to-EXISTS 变换几乎完全一样，但与之相比，MariaDB 10.0 的 FirstMatch 优化有如下优点。

- 在连接多个数据表的情形下优化时，优化器有时会自动添加原查询中没有的相等条件。IN-to-EXISTS 优化中，这种相等条件的传播（Equality propagation）只存在于子查询；而在 FirstMatch 中，由于采用连接形式处理，所以不仅可以传播到子查询，还可以传播到外部查询的数据表。总之，使用 FirstMatch 优化执行时，由于给定更多条件，故可以制定出更好的执行计划。
- IN-to-EXISTS 变换优化策略中，变换无需任何条件时，必定会进行该优化。而 FirstMatch 优化中，可以选择针对子查询的全部数据表进行 FirstMatch 优化，还是只针对部分数据表进行优化。

FirstMatch 优化是特定形态的子查询中经常使用的优化，其限制条件及特征如下所示。

- FirstMatch 优化中，只要从子查询检索到 1 条符合条件的记录就不再继续检索，它是最快捷的路径（Short-cut path）。因此，FirstMatch 优化中先访问子查询引用的所有外部数据表，然后再执行子查询。
- 使用 FirstMatch 优化时，执行计划的 Extra 字段显示 FirstMatch(table-N) 语句。
- FirstMatch 优化也可用于相关子查询。
- FirstMatch 优化无法用于使用 GROUP BY 或聚合函数的子查询优化。

只有 optimizer_switch 的 semijoin 选项与 firstmatch 选项设置为 ON 时，才可以使用 FirstMatch 优化。从 MariaDB 10.0 开始，这两个选项默认处于开启（ON）状态。若只想关闭 firstmatch 优化，则只需将 firstmatch 选项设置为 OFF，semijoin 选项保持默认的 ON 状态即可。

查询带有另外的条件可以使外查询独立使用索引并同时使用子查询时，通常会经常选用 MariaDB 的 FirstMatch 优化。再看看刚才的查询示例。

```
EXPLAIN SELECT *
FROM employees e WHERE e.first_name='Matt'
  AND e.emp_no IN (
    SELECT t.emp_no FROM titles t
    WHERE t.from_date BETWEEN '1995-01-01' AND '1995-01-30'
  );
```

上述示例查询含有“e.first_name='Matt'”比较条件，处理时，外查询可以独立使用索引。执行上述查询时，首先使用该条件将 employees 数据表作为驱动表进行读取，然后使用 FirstMatch 优化读取子查询中使用的 titles 数据表。下面讲解上述查询没有“e.first_name='Matt'”

条件时的情况。

3. Semi-join Materialization 优化

Semi-join Materialization 将半连接中使用的子查询全部具体化，以此进行优化。此处所说的“具体化”（Materialization）简言之就是指创建内部临时表。FirstMatch 优化的最后一部分曾提及下列示例查询。

```
EXPLAIN SELECT *
FROM employees e WHERE e.emp_no IN (
    SELECT t.emp_no FROM titles t
    WHERE t.from_date BETWEEN '1995-01-01' AND '1995-01-30'
  );
```

与 FirstMatch 优化中使用的示例查询相比，上述示例查询只删除了 e.first_name='Matt' 条件。若上述查询使用 FirstMatch 优化，那么关于 employees 数据表的条件就只有子查询，所以必须对 employees 数据表进行全扫描。这种形态的半连接中，使用 FirstMatch 优化对性能提升没有任何帮助。

从 MariaDB 5.3 开始，针对这种形式的查询，使用子查询具体化（Subquery Materialization）优化。从下列执行计划可以看出，上述示例查询的子查询使用了子查询具体化优化。

id	select_type	table	type	key	key_len	ref	rows	Extra
1	PRIMARY	e	ALL	NULL	NULL	NULL	300024	
1	PRIMARY	<subquery2>	eq_ref	distinct_key	4	func	1	
2	MATERIALIZED	t	index	PRIMARY	159	NULL	443308	Using where; Using index

执行计划最后一行的 select_type 字段只显示为“MATERIALIZED”。虽然查询中使用的数据表有 2 个，但是执行计划却显示 3 行，这表示在查询的执行过程中创建了临时表。首先执行读取 titles 数据表的子查询，用执行结果创建临时表（<subquery2>）。然后将 employees 数据表与子查询具体化形成的临时表（<subquery2>）连接起来，最后将结果返回给用户。

上述执行计划中还有一点需要注意，第二行的 key 列显示为 distinct_key。IN(subquery) 形态的查询中，子查询必须只返回唯一值。因此，执行计划中为了保证 emp_no 列值唯一，优化器在具体化后的临时表中自动创建了唯一键。执行计划 key 列显示的 distinct_key 表示执行计划使用了该索引，由此可知，创建的临时表带有唯一索引。

从查询的执行计划可以知道为什么不选择子查询创建的临时表（<subquery2>），而选择需要对更多记录进行全扫描的 employees 数据表作为驱动表。对于子查询具体化时创建的临时表，由于没有固定连接顺序的优化器提示，因而无法先驱动 <subquery2> 临时表。其实，优化器将 employees 数据表选为驱动表是个失误。

为了更有效率地执行针对 titles 数据表的子查询，在 titles 数据表的 from_date 列创建索引，然后查看示例查询的执行计划。

```
ALTER TABLE titles ADD INDEX ix_fromdate(from_date);
```

如上所示创建索引后，创建 <subquery2> 临时表部分的执行计划也变为索引范围扫描。最后，具体化后的 <subquery2> 临时表与 employees 数据表的连接顺序也变得更好。创建索引前后各执行 1 次查询，创建索引后，查询的执行速度变得更快。

id	select_type	table	type	possible_keys	key	Extra
1	PRIMARY	<subquery2>	ALL	distinct_key	NULL	
1	PRIMARY	e	eq_ref	PRIMARY	PRIMARY	
2	MATERIALIZED	t	range	PRIMARY, ix_fromdate	ix_fromdate	Using where; Using index

创建索引后的执行计划中，由于使用具体化后的临时表作为驱动表，所以对创建的临时表进行了全扫描。此时，执行计划第一行的 key 显示为 NULL。由于是对子查询进行的具体化，所以不允许该临时表中的 emp_no 有重复。执行计划的 Extra 列没有显示相关信息，指明具体化后的临时表创建时带有唯一索引。所幸的是，此时执行计划的 possible_keys 列会显示 distinct_key，possible_keys 列平时并没什么用处。也就是说，读取具体化后的临时表时，虽然没有使用任何索引，但其实具体化后的临时表中存在名为 distinct_key 的唯一索引，由优化器决定是否使用。

MATERIALIZED 优化有 Materializtion-Scan 与 Materialization-Lookup 两个策略。示例创建索引之前，具体化后的临时表在连接中用作被驱动表，并且使用 distinct_key 进行检索，这称为 Materialization-Lookup。创建索引之后，具体化后的临时表被用作驱动表，并且进行全表扫描，这称为 Materializtion-Scan。

Semi-join Materializtion 优化与其他子查询优化不同，即使子查询中有 GROUP BY 子句，也可以使用该优化策略。

```
SELECT *
FROM employees e WHERE e.emp_no IN (
    SELECT t.emp_no FROM titles t
    WHERE t.from_date BETWEEN '1995-01-01' AND '1995-01-30'
    GROUP BY t.title
  )
```

上述示例的子查询含有 GROUP BY t.title 子句，但其执行计划仍然可以使用具体化，如下所示。

id	select_type	table	type	possible_keys	key	Extra
1	PRIMARY	<subquery2>	ALL	distinct_key	NULL	
1	PRIMARY	e	eq_ref	PRIMARY	PRIMARY	
2	MATERIALIZED	t	range	PRIMARY,ix_fromdate	ix_fromdate	Using where; Using index

可以使用 Semi-join Materializtion 优化的查询有如下限制条件与特征。

- IN(subquery) 中的子查询必须不是相关子查询。
- 即使使用 GROUP BY 或聚合函数，子查询也可以使用具体化。
- 在使用具体化的情形下使用内部临时表。

只有 optimizer_switch 的 semijoin 选项与 materialization 选项设置为 ON，才可以使用 Semi-join Materializtion 优化。MariaDB 10.0 开始，以上两个选项默认设置为 ON。若想只关闭 Materialization 优化，只要将 semijoin 优化选项设置为 ON、Materializtion 优化选项设置为 OFF 即可。

注意 非半连接的子查询优化也可以使用具体化优化。若 materializtion 优化开关设置为 OFF，则非半连接的子查询优化就无法再使用具体化优化。

4. LooseScan 优化

前面使用索引的 GROUP BY 优化方法中，我们学习过 Using index for group-by 的松散索引扫描，半连接子查询优化的 LooseScan 也采用类似的读取方式。下列简单查询用于读取所有部门号在 dept_emp 数据表中的部门信息。

```
EXPLAIN
SELECT * FROM departments d WHERE d.dept_no IN (
  SELECT de.dept_no FROM dept_emp de
);
```

虽然 departments 数据表中只有 9 条记录，但保存的 departments 数据表的记录数竟然有近 330 000 条。dept_emp 数据表的主键索引由 (dept_no+emp_no) 数据列组合而成。虽然主键全部记录数有 330 000 条左右，但使用 dept_no 数据列分组后，最终只有 9 个分组。那么，采用松散索引方式扫描 dept_emp 数据表的主键，只读取唯一的 dept_no，这样执行子查询部分时会非常高效，而且重复的记录也会同时删除。图 5-28 表示 LooseScan 优化的执行过程。

从图 5-28 中可以看到，子查询中的 dept_emp 数据表用作驱动表，从 dept_no 部分逐条唯一读取 dept_emp 数据表的主键。松散索引扫描的 Using index for group-by 处理方式也与图 5-28 中读取 dept_emp 数据表的主键方式一样。

下列是示例查询的执行计划，在 Extra 列显示为 LooseScan 语句。分配给执行计划各数据表的 id 值都是 1，表示在 MariaDB 内部以类似连接的方式进行处理。

id	select_type	table	type	key	key_len	rows	Extra
1	PRIMARY	de	index	ix_fromdate	3	331603	Using index; LooseScan
1	PRIMARY	d	eq_ref	PRIMARY	12	1	

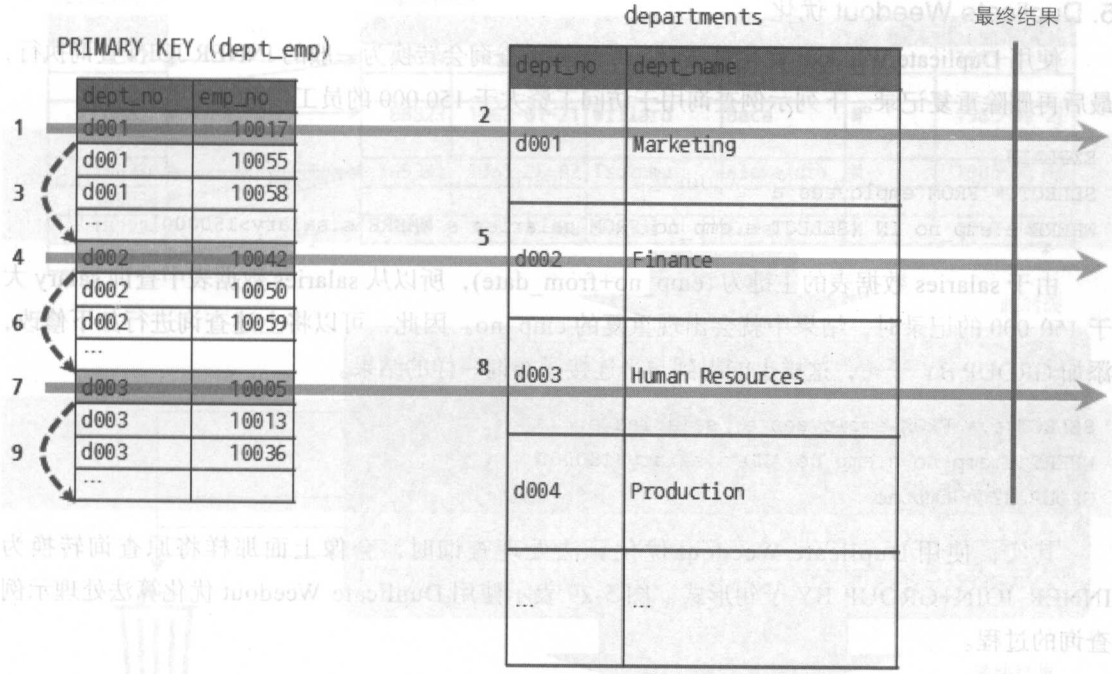


图 5-28 LooseScan 优化工作方式

注意 示例使用的查询非常适合采用 LooseScan 优化方式进行处理。但由于子查询优化还不够成熟，MariaDB 10.0 会尝试使用 Semi-join Materialization 算法执行子查询。若想引导优化器使用 LooseScan 执行计划，需要先临时调整优化器的开关，如下所示。

```
SET optimizer_switch='materialization=off';
SET optimizer_switch='firstmatch=off';
```

我的测试表明，执行该查询时优化器，最先尝试使用具体化优化。将 materialization 优化器开关设置为 OFF 后，又开始使用 FirstMatch 优化制定执行计划。因此，要将 Semi-join Materialization 与 FirstMatch 两个优化选项全部设置为 OFF。

我个人认为最适合该查询的是 FirstMatch 执行计划，松散索引扫描也是比较好的执行计划，但 MariaDB 优化器首先选择了 Semi-join Materialization 优化。

LooseScan 优化有如下特点。

- LooseScan 优化先采用松散索引扫描读取子查询数据表，然后以外部表作为被驱动表进行连接。因此，子查询部分必须具备使用 Loose Index Scan 的条件，才能使用该优化。松散索引扫描优化可以用于以下形式的子查询优化。

```
SELECT .. FROM .. WHERE expr IN (SELECT keypart1 FROM tab WHERE ...)
SELECT .. FROM .. WHERE expr IN (SELECT keypart2 FROM tab WHERE keypart1='常量' ...)
```

- 若想让优化器不使用 LooseScan 优化，只要将 loosescan 优化器开关设置为 OFF 即可。

5. Duplicate Weedout 优化

使用 Duplicate Weedout 优化算法时，半连接子查询会转换为一般的 INNER JOIN 查询执行，最后再删除重复记录。下列示例查询用于访问工资大于 150 000 的员工信息。

```
EXPLAIN
SELECT * FROM employees e
WHERE e.emp_no IN (SELECT s.emp_no FROM salaries s WHERE s.salary>150000);
```

由于 salaries 数据表的主键为 (emp_no+from_date)，所以从 salaries 数据表中查询 salary 大于 150 000 的记录时，结果中就会出现重复的 emp_no。因此，可以将上述查询进行如下修改，添加 GROUP BY 子句，这样也能得到与半连接子查询一样的结果。

```
SELECT e.* FROM employees e, salaries s
WHERE e.emp_no=s.emp_no AND s.salary>150000
GROUP BY e.emp_no
```

其实，使用 Duplicate Weedout 优化算法处理查询时，会像上面那样将原查询转换为 INNER JOIN+GROUP BY 子句形式。图 5-29 表示使用 Duplicate Weedout 优化算法处理示例查询的过程。

(1) 扫描 salaries 数据表的 ix_salary 索引，检索 salary 大于 150 000 的员工，与 employees 数据表进行连接；

(2) 将连接结果存储到临时表；

(3) 从保存在临时表的结果中删除 emp_no 重复的记录；

(4) 删除重复记录后，返回剩余记录。

示例查询使用了 Duplicate Weedout 优化，其执行计划如下所示。虽然执行计划并未单独显示 Duplicate Weedout 这样的语句信息，但 Extra 列显示了 Start temporary 与 End temporary 语句。讲解图 5-29 的处理过程时，将步骤 1 与步骤 2 分开了，其实步骤 1 中的连接操作与步骤 2 中的存储为临时表的处理是反复进行的。反复过程的起始数据表的执行计划行显示 Start temporary 语句，而反复过程结束的数据表的执行计划行显示 End temporary 语句。也就是说，Start/End temporary 语句之间就是 Duplicate Weedout 优化的处理过程。

id	select_type	table	type	key	key_len	ref	rows	Extra
1	PRIMARY	s	range	ix_salary	4	NULL	36	Using where; Using index; Start temporary
1	PRIMARY	e	eq_ref	PRIMARY	4	s.emp_no	1	End temporary

在本书示例的数据表中创建需要使用 Duplicate Weedout 优化处理的查询示例并不容易。因此，在示例查询前先要对 optimizer_switch 选项进行如下调整，禁用 Materialization、FirstMatch、LooseScan 优化。

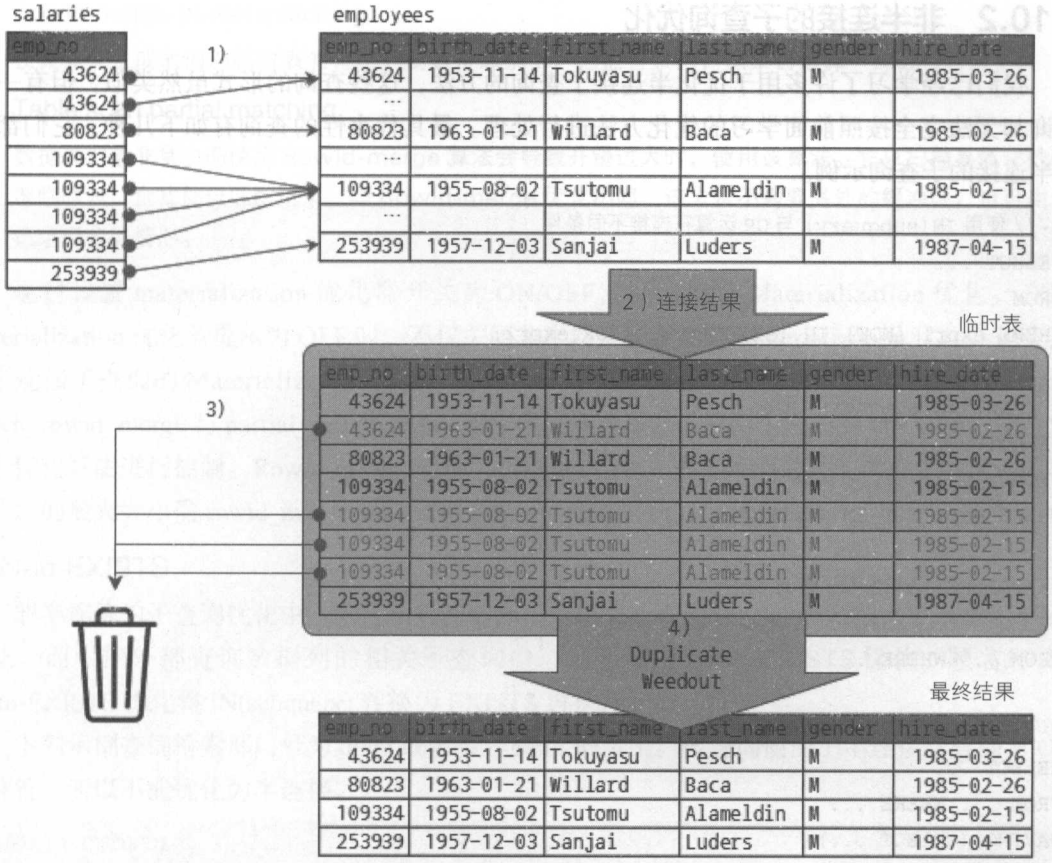


图 5-29 Duplicate Weedout 优化工作方式

```
SET optimizer_switch='materialization=OFF';
SET optimizer_switch='firstmatch=OFF';
SET optimizer_switch='loosescan=OFF';
```

注意 Duplicate Weedout 优化处理方式从“Weed out”的含义可窥一斑。英文词典中对“Weed out”的解释为“清除杂草”。执行 Duplicate Weedout 优化时，先不考虑重复记录进行连接，形成包含重复记录的集合，然后再从中删除重复记录，就像清除杂草一样。该优化方式的命名也非常好，便于用户理解。

Duplicate Weedout 优化有如下限制条件。

- 即使子查询为相关子查询，也能使用 Duplicate Weedout 优化。
- 子查询使用 GROUP BY 或聚合函数时，不能使用。
- 由于 Duplicate Weedout 采用连接方式处理子查询的数据表，所以有很多优化方法。
- 无单独的优化器选项只用于启动或禁用 Duplicate Weedout 优化，借助 semijoin 优化器开关只能启用或关闭半连接子查询的所有优化。

5.10.2 非半连接的子查询优化

我们已经学习了许多用于优化半连接子查询的方法，这些查询的形式虽然类似，但有一些查询却无法完全按照前面学习的优化方法进行处理。最具代表性的查询有如下几种，它们都是非半连接的子查询示例。

```
--// 使用 IN(subquery) 与 OR 运算符连接不同条件
SELECT ...
FROM ...
WHERE expr1 [NOT] IN (SELECT ... ) OR expr2;

--//NOT IN(subquery) 形式
SELECT ...
FROM ...
WHERE expr NOT IN (SELECT ... );

--//SELECT 语句含有子查询
SELECT ..., (SELECT ...)
FROM ... WHERE ...

--//HAVING 语句含有子查询
SELECT ...
FROM ... WHERE ...
HAVING (SELECT ...);

--// 子查询中使用 UNION 时
SELECT ...
FROM ...
WHERE expr IN (SELECT ... UNION SELECT ...);
```

使用这些非半连接的子查询形式时，没有多少可选的优化选项。此情形下，只能使用 Materialization 优化（将子查询具体化为临时表）或 IN-to-EXISTS 优化。

1. Materialization

Materialization 优化与半连接子查询的 Materialization 优化非常类似。Materialization 优化仅用于查询中的子查询不是相关子查询的情形，与半连接子查询的 Materialization 一样，会具体化为临时表。因此，内部临时表的大小受到 tmp_table_size 与 max_heap_table_size 系统设置的影响。

使用 Materialization 优化时，MariaDB 优化器使用如下两种部分匹配算法高效处理 NULL 值。这两部分与半连接子查询的 Materialization 优化是不同的。

Rowld-merge partial matching

在具体化生成的内部临时表的特定列创建特殊索引，扫描各索引，像集合运算一样进行合并。

Table scan partial matching

数据表记录非常少而使用 Rowld-merge 算法会导致开销过大时，使用该算法。它只扫描具体化的内部临时表，查找部分匹配记录。与 Rowld-merge 优化不同，该方法不需要另外的缓冲区，也可用于记录很多的情形。

通过设置 materialization 优化器开关为 ON/OFF，可以控制 Materialization 优化。设置 materialization 优化器提示为 OFF 时，不仅关闭非半连接子查询的 Materialization 优化，也会关闭半连接子查询的 Materialization 优化。此外，针对两种部分匹配算法，分别提供了 partial_match_rowid_merge 与 partial_match_table_scan 优化器选项。设置两个选项为 ON/OFF，能够对两个优化算法进行控制。RowId-merge partial matching 优化在内部使用另外的内存缓冲区，内存缓冲区的最大大小受 rowid_merge_buff_size 系统设置限制。

2. IN-to-EXISTS

非半连接的子查询优化中，只有子查询部分不是相关子查询时，才可以使用 Materialization 优化。而引用外部查询数据列的相关子查询中，只能使用 IN-to-EXISTS 优化。顾名思义，IN-to-EXISTS 优化将 IN(subquery) 转换为 EXISTS 再执行。

下列示例查询检索部门号为 d009 或名字为 Matt 的员工。该查询使用了半连接子查询与 OR 运算符，所以不能优化为半连接。

```
EXPLAIN EXTENDED
SELECT * FROM employees e
WHERE e.emp_no IN (SELECT de.emp_no FROM dept_emp de WHERE de.dept_no='d009')
  OR e.first_name='Matt';
SHOW WARNINGS;
```

该查询的执行计划没什么特别之处，如下所示。

id	select_type	table	type	key	key_len	ref	rows	Extra
1	PRIMARY	e	ALL	NULL	NULL	NULL	300024	Using where
2	DEPENDENT SUBQUERY	de	eq_ref	PRIMARY	16	const,func	1	Using where; Using index

执行 SHOW WARNINGS 命令，从显示的信息可以看到，优化器对查询进行分析并重新组合为分析树的结果，可以发现其内部使用了“<EXISTS>”处理。

```
SELECT 'employees'.'e'.'emp_no' AS 'emp_no',
'employees'.'e'.'birth_date' AS 'birth_date',
'employees'.'e'.'first_name' AS 'first_name',
'employees'.'e'.'last_name' AS 'last_name',
'employees'.'e'.'gender' AS 'gender',
```

```
'employees'.'e'.'hire_date' AS 'hire_date'
FROM 'employees'.'employees' 'e'
WHERE (<expr_cache><'employees'.'e'.'emp_no'>(<in_optimizer>('employees'.'e'.'emp_no',
<EXISTS>(SELECT 'employees'.'de'.'emp_no' FROM 'employees'.'dept_emp' 'de'
        WHERE (('employees'.'de'.'dept_no' = 'd009')
        AND (<CACHE>('employees'.'e'.'emp_no') = 'employees'.'de'.'emp_no')))))
OR ('employees'.'e'.'first_name' = 'Matt'))
```

将 in_to_exists 优化器选项设置为 ON/OFF，可以打开或关闭 IN-to-EXISTS 优化。in_to_exists 优化器选项默认设置为 ON。

5.10.3 子查询缓存

要根据外部查询执行结果中的记录条数反复执行相关子查询时，就会使用子查询缓存优化算法。再看看 IN-to-EXISTS 优化中使用的示例查询。下列示例查询由于不是半连接子查询，所以无法使用半连接优化。

```
EXPLAIN EXTENDED
SELECT * FROM employees e
WHERE e.emp_no IN (SELECT de.emp_no FROM dept_emp de WHERE de.dept_no='d009')
  OR e.first_name='Matt';
SHOW WARNINGS;
```

从上述示例查询的执行计划中可以看到，子查询由 DEPENDENT SUBQUERY 执行计划进行处理。

id	select_type	table	type	Key	key_len	ref	rows	Extra
1	PRIMARY	e	ALL	NULL	NULL	NULL	300024	Using where
2	DEPENDENT SUBQUERY	de	eq_ref	PRIMARY	16	const,func	1	Using where; Using index

查看查询的执行计划后，再使用 SHOW WARNINGS 命令查看优化器重新组合的内容。从优化器重新组合的查询看，IN(subquery) 转换为 EXISTS(subquery) 形式得到执行。

```
SELECT 'employees'.'e'.'emp_no' AS 'emp_no',
'employees'.'e'.'birth_date' AS 'birth_date',
'employees'.'e'.'first_name' AS 'first_name',
'employees'.'e'.'last_name' AS 'last_name',
'employees'.'e'.'gender' AS 'gender',
'employees'.'e'.'hire_date' AS 'hire_date'
FROM 'employees'.'employees' 'e'
WHERE (<expr_cache><'employees'.'e'.'emp_no'>(<in_optimizer>('employees'.'e'.'emp_no',
<EXISTS>(SELECT 'employees'.'de'.'emp_no' FROM 'employees'.'dept_emp' 'de'
        WHERE (('employees'.'de'.'dept_no' = 'd009')
```

```
        AND (<CACHE>('employees'.'e'.'emp_no') = 'employees'.'de'.'emp_no')))))
OR ('employees'.'e'.'first_name' = 'Matt'))
```

employees 数据表为外部表，每读取 1 条记录，EXISTS 的子查询就执行 1 次。此时，将子查询的执行结果放入另外的内存缓存。要执行相同子查询时，不用再次执行子查询，直接将缓存中的内容返回即可。缓存中存储的是用作子查询参数的值（数据列）与子查询的结果，在重新组合的查询内容中，从位于 <expr_cache> 标记后的部分可以知道哪些参数被一起保存到缓存。

参数	<'employees'.'e'.'emp_no'>
缓存结果	(<in_optimizer>('employees'.'e'.'emp_no', <EXISTS>(SELECT 'employees'.'de'.'emp_no' FROM 'employees'.'dept_emp' 'de' WHERE (('employees'.'de'.'dept_no' = 'd009') AND (<CACHE>('employees'.'e'.'emp_no') = 'employees'.'de'.'emp_no')))))

子查询缓存总是与参数值、子查询的结果一起保存到内部临时表。在内部临时表中会将所有保存参数的数据列聚拢在一起，创建唯一索引。对于保存子查询缓存的内部临时表，起初使用内存表（堆表）进行创建，但其变得比 tmp_table_size 或 max_heap_table_size 设定的值大时，就要计算缓存的命中率，决定如何维护缓存。

缓存命中率低时（HitRatio<0.2）

自动禁用子查询缓存。

缓存命中率合理时（0.2<=HitRatio<0.7）

将内部临时表中缓存至今的子查询结果全部删除，重新将子查询结果缓存到内部临时表。

缓存命中率高时（HitRatio>=0.7）

将保存子查询缓存的内部临时表从内存转移到磁盘，继续缓存子查询结果。

第6章 存储引擎

6.1 Aria存储引擎 MariaDB

6.2 XtraDB存储引擎 MariaDB

6.3 InnoDB存储引擎 MariaDB MySQL

6.4 全文检索引擎 MariaDB MySQL

6.5 Memcached插件 MySQL

6.6 Cassandra存储引擎 MariaDB

6.7 CONNECT存储引擎 MariaDB

6.8 Sequence存储引擎 MariaDB

6.9 Mroonga全文检索存储引擎 MariaDB MySQL

针对数据的不同特性与使用形式，MySQL 服务器提供了多种存储引擎。用户可以根据自身需要，以数据表为单位，选用 MEMORY 存储引擎（将数据存储到内存）、MyISAM 或 InnoDB（将数据存储到磁盘）以及其他多种存储引擎。这些存储引擎内部存在差异，比如是否支持事务、锁机制以数据表还是以记录为单位等。但对最终用户而言没什么不同，因为都提供相同访问方式。

与 MySQL 类似，MariaDB 也提供了多种存储引擎。不仅有 MEMORY 存储引擎、MyISAM、InnoDB 储取引擎，还有对这些存储引擎的功能进行扩展的 Aria 存储引擎与 XtraDB 存储引擎。此外，MariaDB 还提供了一些 MySQL 5.6 没有的存储引擎，比如 TokuDB 存储引擎、Cassandra 存储引擎、CONNECT 存储引擎等。限于篇幅，本章无法对 MariaDB 中的所有存储引擎逐一进行讲解，只介绍其中最具代表性的几个存储引擎。

6.1 Aria 存储引擎 MariaDB

MyISAM 存储引擎从 MySQL 初期开始一直使用至今，其功能与结构相对简单，具有处理速度快、容易使用等优点。但由于其不支持事务且只支持表级锁，MySQL 引入 InnoDB 存储引擎后，其使用频率变得越来越低。目前几乎只有内部临时表使用它，或只在非常特殊的条件下才会使用。

为了克服 MyISAM 存储引擎的上述缺点，人们开发了 Aria 存储引擎，它是在 MyISAM 源代码的基础上开发创建的。在 Aria 存储引擎开发早期，它就受到了人们的广泛关注。但随着 XtraDB 与 TokuDB 存储引擎的发展，它正逐渐淡出人们的视线。目前，Monty Program AB 的工作重心放在 MariaDB 服务器本身的改善上，对 Aria 存储引擎的优化工作进展较慢。虽然 Aria 存储引擎还不够成熟，尚无法取代 InnoDB 存储引擎，但它比 MyISAM 存储引擎更高效、更稳定。

MariaDB 需要创建内部临时表时，会使用 Aria 存储引擎而非 MyISAM。因此，了解有关 Aria 存储引擎的基本内容后，对于使用 MariaDB 会非常有帮助。MariaDB 5.5 与 MariaDB 10.0 中的 Aria 存储引擎版本为 1.5，目前版本的 Aria 存储引擎与 MyISAM 最大的不同在于支持事务与页面缓存功能。

6.1.1 事务

创建使用 Aria 存储引擎的数据表时，要明确使用 TRANSACTIONAL 选项，如下所示。仅当创建 Aria 存储引擎数据表时，才使用 TRANSACTIONAL 选项。

```
CREATE TABLE tb_aria (
  fd1 INT NOT NULL,
  fd2 VARCHAR(10) NOT NULL,
  PRIMARY KEY(fd1)
) ENGINE=Aria TRANSACTIONAL=1;
```

如上所示，创建 Aria 存储引擎数据表时打开事务支持功能，这样创建出的数据表就支持事

务。将 TRANSACTIONAL 选项设置为 1 后，像 InnoDB 或 XtraDB 存储引擎一样，相关数据表的所有变更内容就会先保存到日志文件（重做日志），而实际的数据变更内容会延迟处理。令人遗憾的是，MariaDB 10.0 使用的 Aria 存储引擎版本为 1.5，估计 2.0 版本才会加入对事务的完美支持。即便如此，Aria 存储引擎的 TRANSACTIONAL 选项也并非毫无效果。

如果各位有过使用 MyISAM 存储引擎的经验，那肯定会经常遇到数据表结构遭到破坏的情况。这是因为没有使用另外的日志进行自动恢复，如果创建 Aria 存储引擎数据表的同时使用 TRANSACTIONAL 选项，那么 MariaDB 服务器异常退出并重启后，就会利用重做日志进行自动恢复。也就是说，Aria 存储引擎的 TRANSACTIONAL 选项当前主要用于保护数据表，防止数据表发生意外损坏，而非用作事务处理的功能。

6.1.2 页面缓存

与 MyISAM 存储引擎相比，Aria 存储引擎最大的优点是页面缓存。MyISAM 存储引擎通过 key_buffer_size 系统设置可以使用另外的缓存，但该内存缓冲只能缓存索引。Aria 存储引擎与 MyISAM 存储引擎不同，不仅可以缓存索引，甚至数据表的数据页也可以缓存到内存空间。如下所示，创建 Aria 数据表时，将 ROW_FORMAT 选项设置为 page 时，数据文件的内容也可以缓存到内存缓冲。

```
CREATE TABLE tb_aria (
  fd1 INT NOT NULL,
  fd2 VARCHAR(10) NOT NULL,
  PRIMARY KEY(fd1)
) ENGINE=Aria ROW_FORMAT=page;
```

Aria 存储引擎提供了 aria_pagecache_buffer_size 系统变量，通过设置可以调整缓存可用的内存空间大小，默认值为 128 MB。此外，即使不使用 Aria 存储引擎，MariaDB 服务器使用内部临时表处理用户查询时，也会使用 Aria 存储引擎。因此，建议各位将页面缓存最小空间设置为 64 MB~128 MB。

创建使用 Aria 存储引擎的数据表时，若将 ROW_FORMAT 设置为 fixed 而非 page，此时，Aria 存储引擎的工作方式与 MyISAM 几乎一样。也就是说，页面缓存中只缓存索引，数据文件内容存于磁盘或操作系统缓存。

6.1.3 系统变量设置

如同 MyISAM 存储引擎中有多种内存缓冲设置一样，Aria 存储引擎也有额外设置。下面讲解其中几个最具代表性的设置。

aria_pagecache_buffer_size

对于使用 Aria 存储引擎的数据表，该变量用于设置内存空间的大小，以对索引或数据页面进行缓存。

aria_sort_buffer_size

创建使用 Aria 存储引擎的数据表或向 Aria 数据表添加索引时，或者使用 REPAIR 命令恢复使用 Aria 存储引擎的数据表时，将使用内存空间进行排序处理。通过 aria_sort_buffer_size 系统变量可以设置用于排序处理的内存大小。

此处需要注意，对 GROUP BY 或 ORDER BY 进行排序处理时，所用内存空间的大小并非由 aria_sort_buffer_size 变量设定，而由 sort_buffer_size 系统变量进行设定。

aira_group_commit、aria_group_commit_interval

创建 Aria 数据表时，若将 TRANSACTIONAL 选项设置为 1 或 yes，提交所有变更内情时，会先记录到日志（WAL 日志），同时也收集提交的事务日志进行记录。aira_group_commit 设置 Aria 存储引擎是否使用组提交，aria_group_commit_interval 设置收集并记录到磁盘的时间间隔，单位为微秒（1/1000000 秒）。

aria_block_size

对于使用 Aria 存储引擎的数据表，其块大小有 2 KB、4 KB、8 KB 可供选择，默认设置为 8 KB。在大多数服务中都能正常运作，不会有什么问题。

aria_used_for_temp_tables

MariaDB 服务器在内部使用临时表处理用户查询时，将使用 Aria 存储引擎而非 MyISAM 存储引擎。MariaDB 的用户手册指出，内部临时表想使用 MyISAM 存储引擎而非 Aria 存储引擎时，可以通过该系统变量进行设置。但即使重启 MariaDB 服务器，该系统变量设置也不会生效，只有重新编译 MariaDB 服务器才能使其设置起作用。

6.2 XtraDB 存储引擎 MariaDB

XtraDB 是 Percona 公司（http://www.percona.com/）在 InnoDB 源码基础上改进并开发的高性能存储引擎。XtraDB 并不包含于 MySQL 服务器，只在 Percona 开发的 PerconaServer 以及 MariaDB 中使用。

由于 XtraDB 存储引擎是在 InnoDB 源码基础上经过改进开发而成的，所以它们之间具有良好的兼容性，原有的 InnoDB 数据文件可以直接在 XtraDB 存储引擎中使用。如下所示，在 PerconaServer 或 MariaDB 中创建数据表时，内部会创建使用 XtraDB 存储引擎而非 InnoDB 存储引擎的数据表。

```
CREATE TABLE tb_xtradb (
  fd1 INT NOT NULL,
  fd2 VARCHAR(20),
  PRIMARY KEY (fd1)
) ENGINE=InnoDB;
```

开发 XtraDB 存储引擎时，Percona 公司保持了原有 InnoDB 存储引擎的牢固性、事务处理

支持及 MVCC 架构，并使之具有更好的扩展性。经过一系列改善，使用户可以对其进行更精细的调优与监测。Percona 公司 2006 年开始对外针对 MySQL 服务器提供技术支持，拥有丰富的技术支持经验，针对 MySQL 服务器的不足进行了改进提高，开发了 PerconaServer。在性能与功能上较 MySQL 有显著提升，集中体现在 XtraDB 存储引擎上。

MariaDB 服务器与 PerconaServer 中的 XtraDB 版本略有不同。每当推出新版本的 PerconaServer 时，其中包含的 XtraDB 代码即被移植到 MariaDB。有时，PerconaServer 的 XtraDB 拥有的一两项功能无法应用到 MariaDB，但 XtraDB 在 Percona 中的大部分功能可以直接应用于 MariaDB。关于 MariaDB 的 XtraDB 的更详细内容，请参考 PerconaServer 的用户手册（http://www.percona.com/doc/percona-server/5.6/）。

InnoDB 与 XtraDB 存储引擎替换

到 MariaDB 5.5 之前，XtraDB 存储引擎一直用作默认存储引擎。在 MariaDB 5.5 中创建使用 InnoDB 存储引擎的数据表时，默认使用 XtraDB 存储引擎。但从 MariaDB 10.0 开始，XtraDB 不再是默认存储引擎，取而代之的是 Oracle MySQL 的 InnoDB 存储引擎。因此，从 MariaDB 10.0 开始，安装后保持默认设置不变，使用的是 Oracle MySQL 5.6 中的 InnoDB 存储引擎，而非 XtraDB 存储引擎。

若想在 MariaDB 10.0 中将 XtraDB 而非 MySQL 5.6 的 InnoDB 存储引擎用作默认存储引擎，要将 MariaDB 内建的 InnoDB 存储引擎关闭，将 XtraDB 存储引擎模块以插件方式激活。首先安装 MariaDB 服务器，在不进行任何修改的前提下，查看加载的插件目录。使用 SHOW PLUGINS SONAME 命令可以查看加载的插件名称及库文件列表。

```
MariaDB> SHOW PLUGINS SONAME;
+--------------------------+--------------+--------------------+------------+--------+
| Name                     | Status       | Type               | Library    | License|
+--------------------------+--------------+--------------------+------------+--------+
...
| InnoDB                   | ACTIVE       | STORAGE ENGINE     | NULL       | GPL    |
| INNODB_TRX               | ACTIVE       | INFORMATION SCHEMA | NULL       | GPL    |
| INNODB_LOCKS             | ACTIVE       | INFORMATION SCHEMA | NULL       | GPL    |
| INNODB_LOCK_WAITS        | ACTIVE       | INFORMATION SCHEMA | NULL       | GPL    |
| INNODB_CMP               | ACTIVE       | INFORMATION SCHEMA | NULL       | GPL    |
| INNODB_CMP_RESET         | ACTIVE       | INFORMATION SCHEMA | NULL       | GPL    |
| INNODB_CMPMEM            | ACTIVE       | INFORMATION SCHEMA | NULL       | GPL    |
...
| InnoDB                   | NOT INSTALLED| STORAGE ENGINE     | ha_xtradb.so| GPL    |
| XTRADB_READ_VIEW         | NOT INSTALLED| INFORMATION SCHEMA | ha_xtradb.so| GPL    |
| XTRADB_INTERNAL_HASH_TABLES| NOT INSTALLED| INFORMATION SCHEMA | ha_xtradb.so| GPL    |
| XTRADB_RSEG              | NOT INSTALLED| INFORMATION SCHEMA | ha_xtradb.so| GPL    |
```

```
| INNODB_TRX                    | NOT INSTALLED | INFORMATION SCHEMA | ha_xtradb.so | GPL    |
| INNODB_LOCKS                  | NOT INSTALLED | INFORMATION SCHEMA | ha_xtradb.so | GPL    |
| INNODB_LOCK_WAITS             | NOT INSTALLED | INFORMATION SCHEMA | ha_xtradb.so | GPL    |
| INNODB_CMP                    | NOT INSTALLED | INFORMATION SCHEMA | ha_xtradb.so | GPL    |
| INNODB_CMP_RESET              | NOT INSTALLED | INFORMATION SCHEMA | ha_xtradb.so | GPL    |
| INNODB_CMPMEM                 | NOT INSTALLED | INFORMATION SCHEMA | ha_xtradb.so | GPL    |
...
+-------------------------------+---------------+--------------------+--------------+--------+
```

从上述结果看，Name 字段值为“InnoDB”的共有两项，一个 Library 字段为 NULL，另一个为 ha_xtradb.so。Library 字段为 NULL 的 InnoDB 是内建于 MariaDB 中的默认 InnoDB 存储引擎，ha_xtradb.so 库是 XtraDB 存储引擎。内建 InnoDB 存储引擎的 Library 字段值之所以为 NULL，是因为 InnoDB 存储引擎并非以插件形式创建，而是静态包含于 MariaDB 服务器的 mysqld 程序。由 Status 字段值可知，并未安装 ha_xtradb.so 库。

注意 经过我的测试，到 MariaDB 10.0.8 为止，安装在 Windows 中的 MariaDB 服务器并未提供 XtraDB 存储引擎。目前，Percona 好像也没有开发 Windows 版本的 PerconaServer(XtraDB) 的计划。书中给出的是在 MariaDB 10.0.6 版本中运行的示例，与在 MariaDB 10.0.9（XtraDB 为默认内置存储引擎）中运行的结果可能略有不同。此外，下面要讲到的 ignore_buildin_innodb 与 plug-in 选项在 MariaDB 10.0.9 以上版本中（XtraDB 为默认内置存储引擎）也不是必要的。

下面修改 MariaDB 服务器的设置文件，关闭内建的 InnoDB 存储引擎，并加载 XtraDB 存储引擎，如下所示。在 MariaDB 服务器的 my.cnf 设置文件中添加如下内容，然后重启 MariaDB 服务器，使修改生效。

```
[mysqld]
ignore_builtin_innodb = ON
plugin-load = ha_xtradb.so
```

下面重启 MariaDB 服务器，再次使用 SHOW PLUGINS SONAME 命令，查看加载到 MariaDB 服务器的插件库列表。

```
MariaDB> SHOW PLUGINS SONAME;
+-------------------------+--------------+--------------------+------------+--------+
| Name                    | Status       | Type               | Library    | License|
+-------------------------+--------------+--------------------+------------+--------+

| InnoDB                  | NOT INSTALLED | STORAGE ENGINE     | NULL       | GPL    |
| INNODB_TRX              | NOT INSTALLED | INFORMATION SCHEMA | NULL       | GPL    |
| INNODB_LOCKS            | NOT INSTALLED | INFORMATION SCHEMA | NULL       | GPL    |
| INNODB_LOCK_WAITS       | NOT INSTALLED | INFORMATION SCHEMA | NULL       | GPL    |
| INNODB_CMP              | NOT INSTALLED | INFORMATION SCHEMA | NULL       | GPL    |
| INNODB_CMP_RESET        | NOT INSTALLED | INFORMATION SCHEMA | NULL       | GPL    |
```

```
...
| InnoDB                        | ACTIVE   | STORAGE ENGINE     | ha_xtradb.so | GPL  |
| XTRADB_READ_VIEW              | ACTIVE   | INFORMATION SCHEMA | ha_xtradb.so | GPL  |
| XTRADB_INTERNAL_HASH_TABLES   | ACTIVE   | INFORMATION SCHEMA | ha_xtradb.so | GPL  |
| XTRADB_RSEG                   | ACTIVE   | INFORMATION SCHEMA | ha_xtradb.so | GPL  |
| INNODB_TRX                    | ACTIVE   | INFORMATION SCHEMA | ha_xtradb.so | GPL  |
| INNODB_LOCKS                  | ACTIVE   | INFORMATION SCHEMA | ha_xtradb.so | GPL  |
| INNODB_LOCK_WAITS             | ACTIVE   | INFORMATION SCHEMA | ha_xtradb.so | GPL  |
| INNODB_CMP                    | ACTIVE   | INFORMATION SCHEMA | ha_xtradb.so | GPL  |
| INNODB_CMP_RESET              | ACTIVE   | INFORMATION SCHEMA | ha_xtradb.so | GPL  |
| INNODB_CMPMEM                 | ACTIVE   | INFORMATION SCHEMA | ha_xtradb.so | GPL  |
...
+-------------------------------+----------+--------------------+--------------+-------+
```

从上述结果可以看到，MySQL 5.6 的 InnoDB 存储引擎处于禁用状态（NOT INSTALLED），XtraDB 存储引擎处于激活状态（ACTIVE），所显示的名称为 InnoDB。使用 CREATE TABLE .. ENGINE=InnoDB 语句创建数据表时，将创建出使用 XtraDB 存储引擎的数据表。

XtraDB 存储引擎是在 InnoDB 存储引擎源代码的基础上开发而成的，它们之间具有良好的兼容性，几乎所有数据文件都可以通用，并且其功能也都类似。其实，XtraDB 存储引擎在 InnoDB 存储引擎原有功能基础上又添加了额外功能，所以 XtraDB 是 InnoDB 的超集。关于 XtraDB 存储引擎更详细的介绍，请与 InnoDB 存储引擎的介绍进行对比学习，这将有利于各位进一步理解。对于 XtraDB 存储引擎的介绍到此结束，在 InnoDB 存储引擎部分将进一步学习有关 XtraDB 与 InnoDB 的内容。

6.3 InnoDB 存储引擎 MariaDB MySQL

从 MySQL 5.5 升级到 MySQL 5.6 后，InnoDB 存储引擎也发生了许多变化。正如前面讲解 XtraDB 存储引擎时提及的，XtraDB 与 InnoDB 拥有相当多的共同之处。下面将先介绍 MySQL 5.6 中的 InnoDB 的功能，然后再学习有关 MariaDB 10.0 的 XtraDB 存储引擎的内容。学习时不必费心思地区分某功能到底是 InnoDB 的还是 XtraDB 的，由于 XtraDB 拥有 InnoDB 的全部功能，本部分涉及的所有功能说明同样适用于 MariaDB 的 XtraDB 拥有的功能。

6.3.1 MySQL 5.6 InnoDB

XtraDB 存储引擎保留了 MySQL 的 InnoDB 存储引擎的所有功能，并添加了一些新功能，对原有功能进行了扩展与改善。因此，若要理解 XtraDB 存储引擎，必须先掌握 MySQL 内置的 InnoDB 存储引擎的功能。下面对 MySQL 5.6 的 InnoDB 存储引擎的新功能进行简单介绍。

1. 永久性统计信息

与之前版本不同，MySQL 5.6 的 InnoDB 采用数据表分别管理各数据表的统计信息。MySQL 5.5 及其以前版本的 InnoDB 中，统计信息由各存储引擎在内存中进行管理，经常发生变更，以至于主从服务器分别制定不同的查询执行计划。因此，MySQL 5.6 的 InnoDB 中做了进一步改善，根据数据表的全部记录数或索引，分别使用 mysql 数据库的 innodb_index_stats 与 innodb_table_stats 数据表管理基数（Cardinality）信息。并且引入了 innodb_stats_auto_recalc 系统变量，允许用户决定是否自动变更该信息。这样将主服务器的统计信息复制到从服务器，进而防止主从服务器执行计划出现差异。

但 MySQL 5.6 的 InnoDB 中，并未提供更多管理数据表统计信息的功能，只将存储地点从内存转移到基于磁盘的数据表。前面学习执行计划时已经提到，与存储引擎种类无关，MariaDB 10.0 从非存储引擎级别的 MariaDB 服务器角度，使用 mysql 数据库中的 table_stats、index_stats、column_stats 数据表管理统计信息。MariaDB 10.0 不仅可以管理无索引数据列的统计信息，甚至还可以管理数据列的直方图。

2. 数据读取优化

针对 MySQL 5.6 的索引条件下推与多范围读优化，MySQL 服务器的 Handler API 得到了改善。为了使这些功能正常运行，各存储引擎必须全部实现这些 Handler API。MySQL 中常用的 InnoDB 与 MyISAM 存储引擎全都实现了这些 API，所以可以使用索引条件下推与多范围读功能。但使用 TokuDB 或 Mroonga 等第三方存储引擎时，要认真阅读使用手册，查看其是否支持这些功能。

3. 内核互斥体

InnoDB 内部拥有许多共享内存对象，最具代表性的有缓冲池的各个块以及重做日志等。处理查询时，这些共享的内存对象由众多客户连接竞争占用，然后释放。对于这些内存对象，必须要进行同步处理，以使各客户连接彼此不能同时占有。为实现这一目的，就要使用锁（互斥体或信号量）。

但并非所有共享内存对象都拥有各自的锁，否则会有大量锁被混用，锁的管理就会变得更为复杂。有时在经常发生锁竞争的部分，使用一个锁处理多个内存结构体的同步，经常会发生不必要的锁等待情形。其中最具代表性的锁为 kernel_mutex 互斥体。仅用三言两语不能阐明 kernel_mutex 的用途。MySQL 源代码中无符合用途的互斥体或信号量时，kernel_mutex 可以用作通用互斥体。采用这种方式使用 kernel_mutex 时，即使不会对 InnoDB 的性能产生太大影响，kernel_mutex 也会成为瓶颈，变为妨碍扩展性的因素。

MySQL 5.6 的 InnoDB 将滥用的 kernel_mutex 根据不同用途进一步细分，引入了特别的锁。最具代表性的改进之处是，事务的并发控制与 MVCC 等内存结构体都由单独的读写锁进行控制。

4. 基于多线程的 Undo 清理

为了进行 MVCC（Multi Version Concurrency Control，多版本并发控制）与回滚，InnoDB 单独管理着 Undo 空间（Undo Space、Rollback segment）。假设用户执行如下 UPDATE 语句。

```
--// 假设 member 数据表中存储着如下记录。
INSERT INTO member(m_id, m_name, m_area) VALUES (12, '洪吉童', '首尔');
--// 然后执行如下 UPDATE 语句。
UPDATE member SET m_area='京畿' WHERE m_id=12;
```

执行上述 UPDATE 语句后，即使不执行 COMMIT，member 数据表列也会立即修改为“京畿”。但是，InnoDB 存储引擎无法得知用户执行 UPDATE 语句后执行 ROLLBACK 操作还是 COMMIT 操作。因此，为了应对用户可能进行 ROLLBACK 操作的情形，先要将修改为“京畿”之前的值（“首尔”）保存到某个地方。

图 6-1 表示执行 UPDATE 语句后，进行 COMMIT 或 ROLLBACK 之前的 InnoDB 缓冲池与磁盘数据文件的状态。此状态下，若用户进行了 COMMIT，缓冲池中变更的内容就会得到永久适用；若进行 ROLLBACK 操作，就会获取 Undo 日志中的“首尔”信息，将 InnoDB 缓冲池中的记录信息复原。

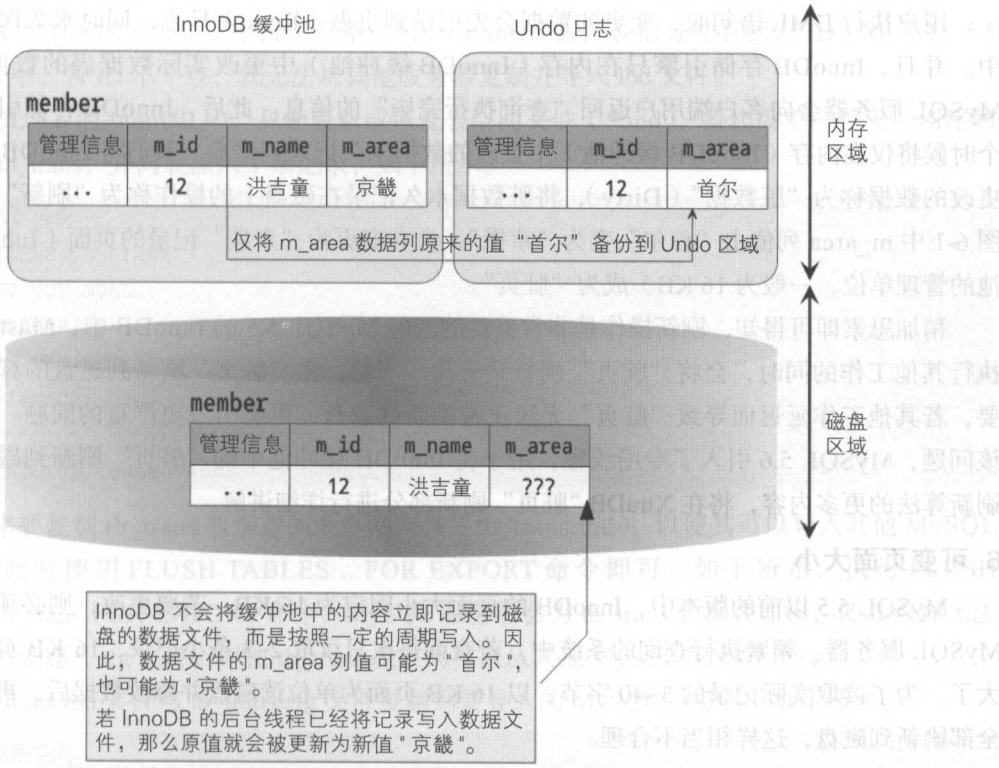


图 6-1 UPDATE 后 InnoDB 缓冲池与数据文件及 Undo 区域的变化

许多客户连接中的数据发生变化时，Undo 日志区域中就会累积很多变更前的信息。对于这些累积的 Undo 日志必须在某个时候进行删除，留出足够的空间，这样才能继续保存以后的变更内容。删除 Undo 日志的操作称为 Undo 清理（Undo purge）。MySQL 5.1 以前版本中，Undo 清理工作由 InnoDB 主线程负责处理。然而，由于 InnoDB 的 Master Thread 还承担了 Undo 清理之外的许多工作，这可能导致 Undo 清理无法正常进行，或者造成 Master Thread 其他工作发生延迟现象。

MySQL 5.5 的 InnoDB 引入了专门负责 Undo 清理的线程，在此之前，Undo 清理工作只由单一线程进行。通过 MySQL 服务器的 innodb_purge_threads 系统变量可以设置 Undo 清理线程的个数，MySQL 5.5 之前，其值只能设置为 0 或 1。将 innodb_purge_threads 设置为 0 时，不使用专用的 Undo 清理线程。MySQL 5.1 中，由 InnoDB 的 Master Thread 进行 Undo 清理工作；若设置为 1，则会激活一个专用的 Undo 清理线程。但有时这样也不够，MySQL 服务器中发生大量 UPDATE 操作时，Undo 日志就会一直增加。

MySQL 5.6 的 InnoDB 中，允许用户将 innodb_purge_threads 系统变量设置为大于 1 的值，这样可以同时使用多个 Undo 清理线程进行清理工作。

5. 独立的刷新线程

用户执行 DML 语句时，变更的数据会先记录到重做（Redo）日志，同时永久保留在磁盘中。并且，InnoDB 存储引擎只在内存（InnoDB 缓冲池）中更改实际数据表的数据。至此，MySQL 服务器会向客户端用户返回“查询执行完毕”的信息。此后，InnoDB 存储引擎会在某个时候将仅在内存（InnoDB 缓冲池）中更改的数据永久记录到磁盘。我们将 InnoDB 缓冲池中更改的数据称为“脏数据”（Dirty），将脏数据永久记录在磁盘上的操作称为“刷新”（Flush）。图 6-1 中 m_area 列值由“首尔”变为“京畿”，含有变更为“京畿”记录的页面（InnoDB 缓冲池的管理单位，一般为 16 KB）成为“脏页”。

稍加思索即可得知，刷新操作是非常重要的。但 MySQL 5.5 的 InnoDB 中，Master Thread 执行其他工作的同时，会将“脏页”刷新到磁盘。当然，将“脏页”刷新到磁盘的算法也很重要，若其他工作延迟而导致“脏页”无法正常刷新到磁盘，可能引发更严重的问题。为了解决该问题，MySQL 5.6 引入了专用线程，用于将 InnoDB 缓冲池中的“脏页”刷新到磁盘。关于刷新算法的更多内容，将在 XtraDB“脏页”刷新部分进行详细讲解。

6. 可变页面大小

MySQL 5.5 以前的版本中，InnoDB 的页面大小固定为 16 KB。若要更改，则必须重新编译 MySQL 服务器。频繁执行查询的系统中，若查询每次只读取 2~3 条小记录，16 KB 页面显然太大了。为了读取实际记录的 3~40 字节，以 16 KB 页面为单位读磁盘并修改数据后，再将 16 KB 全部刷新到磁盘，这样相当不合理。

MySQL 5.6 的 InnoDB 存储引擎中，可以将数据或索引页面的大小从原有的 16 KB 调整为 4 KB 或 8 KB。若 MySQL 服务器频繁执行的查询只是通过主键或索引简单地读取 1~2 条记录，

那么此时原有的 16 KB 页面大小就不太合适，4 KB 左右的页面大小将使工作更有效率。并且，4 KB 页面大小在近来常用的固态硬盘（Solid State Drive，SSD）中的工作效率也非常高。普通硬盘中，移动磁头到读取数据的位置要花费很多时间，而从相应位置开始连续读取多少数据显得并不怎么重要。也就是说，从特定位置开始，要连续读取的数据是 4 KB 还是 16 KB 并不会对性能产生影响。但对于 SSD 这类无磁头的硬盘，查找要读取数据的位置几乎不花费时间，而读取并传送数据的工作会对性能产生影响。不使用普通 HDD 而使用 SSD 时，将 InnoDB 的页面大小降低为 4 KB 也是非常有用的。

注意 已经使用 16 KB 的页面大小创建了系统表空间，且也存在 16 KB 大小的用户数据表时，必须转储 MySQL 服务器的所有数据，然后再进行加载。此时，要使用 mysqldump 命令或 SELECT INTO .. OUTFILE .. 等逻辑备份命令。

XtraBackup 或 Oracle Enterprise backup 等工具会原样复制所有数据文件，即使再次恢复备份，数据文件仍然会像以前一样使用 16 KB 大小的页面。

7. 表空间复制

MySQL 5.5 的 InnoDB 也提供了复制表空间的功能。但 MySQL 5.5 提供的该功能只能复制从自身的 MySQL 服务器备份的表空间文件（*.ibd）。InnoDB 存储引擎内部的字典信息包含所有 InnoDB 数据表的元信息，字段信息中，每个 InnoDB 数据表都分配有固定 ID。MySQL 5.5 的 InnoDB 中，若 ID 不一致，就无法从其他服务器复制并导入 ibd 文件。

MySQL 5.6 的 InnoDB 中，只能获取或恢复表空间文件（*.ibd），操作简单。首先，创建测试数据表 tb_trans，并向表插入 1 条记录，如下所示。

```
--// 示例数据表
CREATE TABLE tb_trans (
  fd1 INT NOT NULL,
  fd2 VARCHAR(20),
  PRIMARY KEY(fd1)
)ENGINE=InnoDB;

INSERT INTO tb_trans VALUES (9, 'Matt');
```

接下来要复制 tb_trans 数据表的表空间文件（tb_trans.ibd），以使其可以导入其他 MySQL 服务器，此时使用 FLUSH TABLES .. FOR EXPORT 命令即可。如下所示，执行 FLUSH TABLES tb_trans FOR EXPORT 命令，MySQL 服务器会为 tb_trans 数据表加锁，使其内容无法更改，然后创建 *.cfg 文件，以使 tb_trans 表空间可以从其他 MySQL 服务器导入。

将 tb_trans.ibd 文件与 tb_trans.cfg 文件复制到要导入的 MySQL 服务器，如下所示。

```
--// 导出表空间
mysql> FLUSH TABLES tb_trans FOR EXPORT;
```

```
--// 复制导出的表空间文件
shell> cp /data/test/tb_trans.ibd /data/test/tb_trans.cfg  /backup/

--// 解除数据表锁
mysql> UNLOCK TABLES;
```

执行 FLUSH TABLES .. FOR EXPORT 命令，相关数据表就会加上明锁，其他用户可以读该数据表中的记录，但无法进行修改。若需要，导出文件复制完成后，可以立即使用 UNLOCK TABLES 命令对数据表进行解锁。

接下来，在要导入复制的 tb_trans.ibd 文件的 MySQL 服务器中，使用 ALTER TABLE .. DISCARD TABLESPACE 命令删除原数据表的表空间。执行 ALTER TABLE .. DISCARD TABLESPACE 命令时，MySQL 服务器的原有数据表空间文件会被删除，请各位注意。ALTER TABLE tb_trans DISCARD TABLESPACE 命令执行完毕后，再将导出并复制的表空间文件 tb_trans.ibd 与 tb_trans.cfg 文件复制到 tb_trans.FRM 文件所在目录。若有必要，请适当修改权限。

```
--// 导入表空间
mysql> ALTER TABLE tb_trans DISCARD TABLESPACE;

shell> cp /backup/tb_trans.cfg /backup/tb_trans.ibd /data/test/

mysql> ALTER TABLE tb_trans IMPORT TABLESPACE;

--// 查看数据
mysql> SELECT * FROM tb_trans;
+-----+------+
| fd1 | fd2  |
+-----+------+
|   9 | Matt |
+-----+------+
```

表空间文件复制到指定位置后，再使用 ALTER TABLE tb_trans IMPORT TABLESPACE 命令将复制的 ibd 文件导入 MySQL 服务器即可。导出表空间再进行导入时，虽然各数据表的名称可以不同，但数据表的结构与所用的存储引擎必须相同。像这样，将表空间全部导出再导入可以在 MySQL 服务器之间快速复制大的数据文件。对 MySQL 服务器进行运维时，该功能是非常有用的，请各位牢记。

提示 若想将从其他 MySQL 服务器获取的 InnoDB 表空间文件（*.ibd）导入原有的 InnoDB 数据表，首先要使用 ALTER TABLE .. DISCARD TABLESPACE 命令删除原数据表的表空间文件（*.ibd）。此时执行 ALTER TABLE .. DISCARD TABLESPACE 命令，InnoDB 存储引擎就会直接从磁盘删除该文件。

若要为了备份或特殊目的而保留原 InnoDB 数据表的表空间文件，在 Unix 系列的操作系统中，如下使用硬链接即可。

```
## 利用 Unix 系列的硬链接将原数据表的数据文件链接到备份区域。
## ln 命令中不可使用 -s 选项，-s 选项用于创建软链接而非硬链接。
shell> ln $DATA_DIR/db_test/old_table.ibd  $BACKUP_DIR/old_table.ibd

MySQL> ALTER TABLE old_table DISCARD TABLESPACE;
MySQL> ALTER TABLE old_table IMPORT TABLESPACE;
```

Unix 系列操作系统中，InnoDB 存储引擎删除数据文件或其他文件时，会调用 unlink 内核函数（系统调用）。对于创建有硬链接的文件，调用 unlink 内核函数只会将元信息的链接数减少 1，并不会删除实际的数据文件。也就是说，上述示例中，实际删除的对象文件“$DATA_DIR/db_test/old_table.ibd”虽然已经从相关目录中消失，但备份目录中创建的硬链接文件“$BACKUP_DIR/old_table.ibd”仍然原样保留。当然，若要使用 unlink 内核函数删除的文件只有 1 个硬链接，则该文件中的数据也会从磁盘中删除。

特定文件系统中，删除大容量文件需要进行相当多的写操作，所以删除大容量文件可能引起服务查询速度减慢。使用 ALTER TABLE .. DISCARD TABLESPACE 命令也一样，删除数据文件也可能产生不良影响。此情形下，在单独的备份目录中创建好硬链接，再执行 ALTER TABLE .. DISCARD TABLESPACE 命令，InnoDB 存储引擎会删除相关数据文件。但 Linux 操作系统中只会断开链接，而不会删除实际文件，所以丢弃（DISCARD）表空间时不会对服务产生太大的不良影响。

8. 独立的 Undo 空间

MySQL 5.5 之前的 InnoDB 中，Undo 区域是系统表空间的一部分。但插入缓冲或 DoubleWrite 缓冲等数据也会使用系统表空间。问题在于，Undo 区域主要基于随机 I/O 工作，而 DoubleWrite 缓冲基于顺序 I/O 向磁盘写记录。这意味着，很难为保存系统表空间而选定磁盘位置。

然而，MySQL 5.6 的 InnoDB 中引入了如下 3 个系统变量，用于将 Undo 区域放入非系统表空间的单独空间。

innodb_undo_directory

设置要存储 Undo 区域（Undo space、Undo tablespace，该区域保存 InnoDB 的 Undo 日志）的目录。innodb_undo_directory 的默认设置为“.”，含义与 MySQL 5.5 中一样，表示 Undo 区域使用系统表空间。

innodb_undo_tablespaces

也可以将 Undo 区域想象为一个数据表，像分割普通数据表的记录一样，创建 Undo 区域时也可以划分为多个数据表空间。Undo 数据表空间最多可以创建 126 个，若将 innodb_undo_tablespaces 系统变量设置为 6，就会在 innodb_undo_directory 设定的 Undo 目录中生成 6 个以 undo 为前缀的文件。若对 Undo 区域的使用竞争激烈，可增大 innodb_undo_tablespaces 设置值，以减少对 Undo 段的互斥体竞争。

innodb_undo_logs

该系统设置指定回退段（Rollback segment）的个数，InnoDB 中最多可有 1023 个写事务共享 1 个回退段。innodb_undo_logs 变量的最大值可设为 128。比如，将 innodb_undo_logs 的值设为 20，则

同时最多可以执行 20 460（1023*20）个写事务。增加回退段的个数后，由于无法减少不创建新数据库的异常，所以建议刚开始时要将 innodb_undo_logs 值设置得小一些。回退段竞争激烈时，再将其设置得大一些。

若将 innodb_undo_directory 设置的 Undo 区域与系统表空间单独分开，最好把 Undo 区域设置到速度尽可能快的磁盘卷。经过进一步改善，使 Undo 区域可以使用单独的目录，其实这也是基于使用 SSD 等快速存储媒介进行的考虑。

9. 只读事务优化

MySQL 5.5 以前的 InnoDB 存储引擎中，处理所有查询时都会自动开始 / 结束事务。这与开启 Auto-Commit 的情形是一样的。为了保证事务正常进行，内部要派发事务 ID，并且每次都要分配内存结构体，以保持事务正常运行。

MySQL 5.6 的 InnoDB 中，对于下列两种情况，将使用只读事务（Read-only transaction）进行处理。

- 以“START TRANSACTION READ ONLY”开始的事务
- AutoCommit 处于激活状态下而只进行 SELECT 查询的事务（SELECT .. FOR UPDATE 除外）

对于满足以上两个条件的事务，InnoDB 存储引擎会将其识别为只读事务，不会派发事务 ID，也不会分配变更数据所需的内存结构体。事务 ID 减少时，所有连接执行查询或 DML 语句时，要引用并检索的内存结构体变得比较简单，所以能获得更快的执行性能。

以“START TRANSACTION READ ONLY”开始的只读事务中，执行变更数据的 UPDATE 或 INSERT 等语句时，将会引发如下错误。

```
ERROR 1792 (25006): Cannot execute statement in a READ ONLY transaction.
```

此外，由于只读事务信息被单独管理，所以执行 SHOW ENGINE INNODB STATUS 命令将不会输出只读事务信息。只读事务信息只能通过 information_schema 数据库的 INNODB_TRX 数据表进行引用。INNODB_TRX 数据表中存在名为 trx_is_read_only 的数据列，借助该数据列可以区分是否为只读事务。

10. 缓冲池转储与加载

使用 MySQL 5.5 时，MySQL 服务器重启后，刚开始的 2~30 分钟内最让人紧张，必须时刻监测。许多有经验的读者都知道，MySQL 服务器重启时，MySQL 5.5 的 InnoDB 中，缓冲池起初空空如也。

经过长时间的查询处理后，InnoDB 会将常用数据表全部加载到缓冲池。因此，大部分查询都能得到快速处理，其他少数查询从磁盘读取数据即可。MySQL 服务器重启时，内存将不会保留任何数据，所以所有查询都要读磁盘。这种现象非常严重时，经常会导致无法启动服务。使用 MySQL 5.5 时，每次重启 MySQL 服务器都要对数据表与索引进行全扫描，强制进行“热身运动”。但由于这种方式与使用频率无关，即使将数据加载到 InnoDB 的缓冲池，通常也不会有

太大作用。当然，MariaDB 从 5.5 版本开始支持缓冲池的数据转储与加载功能，使用这些功能可以轻松解决上述问题。

(1) MariaDB 5.5 的缓冲池转储与加载 MariaDB

MariaDB 5.5 与 PerconaServer 5.5 的 XtraDB 都包含了 MySQL 5.6 中新增的缓冲池预热功能。在 MariaDB 5.5 中使用 innodb_auto_lru_dump 系统变量后，可以在 MariaDB 服务器终止时将 XtraDB 缓冲池中的全部内容转储为文件。MariaDB 服务器启动时，再将文件形式的缓冲池信息读入 XtraDB 的缓冲池，该过程是自动进行的。只要将 innodb_auto_lru_dump 系统变量设置为非 0 值，就能自动进行缓冲池转储与加载操作。

此外，也可以使用 SQL 命令，在需要的时候将 XtraDB 缓冲池中的内容转储，然后再进行加载操作。MariaDB 5.5 中的缓冲池转储与加载功能不需要使用另外的 SQL 语法，只需通过 information_schema 数据库的 XTRADB_ADMIN_COMMAND 数据表即可将 XtraDB 的缓冲池内容转储为文件，然后再进行加载应用。下面通过简单示例进行讲解。

```
MariaDB> SELECT * FROM information_schema.XTRADB_ADMIN_COMMAND /*!XTRA_LRU_DUMP*/;
+------------------------------+
| result_message               |
+------------------------------+
| XTRA_LRU_DUMP was succeeded. |
+------------------------------+
```

执行以上命令，XtraDB 就会将缓冲池的数据页目录（LRU 链表）以 ib_lru_dump 文件形式记录到 MariaDB 服务器的数据目录。再次将转储到 ib_lru_dump 文件的缓冲池内容加载至 XtraDB 的缓冲池时，只要对 information_schema.XTRADB_ADMIN_COMMAND 数据表执行 SELECT 查询即可，如下所示。

```
MariaDB> SELECT * FROM information_schema.XTRADB_ADMIN_COMMAND /*!XTRA_LRU_RESTORE*/;
+---------------------------------+
| result_message                  |
+---------------------------------+
| XTRA_LRU_RESTORE was succeeded. |
+---------------------------------+
```

将 XtraDB 缓冲池转储到 ib_lru_dump 文件并不意味着缓冲池中的所有数据都会被记录到 ib_lru_dump 文件。也就是说，若 XtraDB 缓冲池为 100 GB，则 ib_lru_dump 文件要小于 100 GB。执行转储 XtraDB 缓冲池的命令后，XtraDB 存储引擎只会收集缓冲池所有缓存页面的 space_id 与 page_no 值，并依次对 space_id 与 page_no 进行排序，再记录到 ib_lru_dump 文件。之所以先排序再存储是因为，以后需要从磁盘读取相关数据表的数据页时，可以采用顺序 I/O 方式（非随机 I/O 方式）一次读取大量数据，从而大大加快加载速度。

space_id 的值用于识别数据表空间，而 page_no 值用于在一个数据表空间中识别特定页面

(block 块)。由于两个值都是 8 字节的常量，所以每个页面会有 16 字节被记录到 ib_lru_dump 文件。若所用缓冲池的大小为 100 GB，则记录到 ib_lru_dump 文件中的内容最多为 100 MB ((10010241024/16)*(8+8))，具体的量会根据数据表的个数或模型的特性而有所不同。实际上，100 GB 大小的缓冲池对应生成的 ib_lru_dump 文件大约为 50 MB，20 GB 的缓冲池大约生成 1~5 MB 的 ib_lru_dump 文件。

其实，即使转储 100 GB 大小的缓冲池，也不会花费很长时间，大约为 1~2 秒。但加载时间会根据数据特点可能需要花费很长时间，此时查看 innodb_buffer_pool_pages_data 状态变量即可获知当前缓冲池的加载程度。

```
MariaDB> SHOW STATUS LIKE 'Innodb_buffer_pool_pages_data';
+---------------------------------+---------+
| Variable_name                   | Value   |
+---------------------------------+---------+
| innodb_buffer_pool_pages_data | 352219  |
+---------------------------------+---------+
```

此外，通过计算与 innodb_buffer_pool_pages_total 状态变量的比值，可以得知加载进度的百分比。

```
MariaDB> SHOW STATUS LIKE 'Innodb_buffer_pool_pages_total';
+---------------------------------+---------+
| Variable_name                   | Value   |
+---------------------------------+---------+
| innodb_buffer_pool_pages_total | 2047152 |
+---------------------------------+---------+

MariaDB> SELECT 352219*100/2047152 as loaded_pct;
--> 17.20
```

将转储的缓冲池数据再次加载到 MariaDB 服务器时，MariaDB 仍然会接收来自用户的查询请求。缓冲池的数据加载很慢时，来自用户的查询请求就会处理得相当慢。因此，在 MariaDB 中可以使用 innodb_blocking_buffer_pool_restore 系统变量，对缓冲池进行预热（加载）期间，禁止执行使用 XtraDB 存储引擎数据表的查询。将 innodb_blocking_buffer_pool_restore 系统变量设置为 ON 后，加载缓冲池过程中就不会再执行用户查询。innodb_blocking_buffer_pool_restore 系统变量的默认值为 OFF，不阻止用户查询。

注意 从 MySQL 5.6 开始，Oracle 的 InnoDB 本身提供对缓冲池的元信息进行转储与加载的功能。因此，从 MariaDB 10.0 开始，不再提供使用 MariaDB 5.5 的 XTRADB_ADMIN_COMMAND 数据表的缓冲池进行转储与加载的命令。使用 MariaDB 10.0 时，必须使用 MySQL 5.6 中的缓冲池转储与加载命令。

(2) MySQL 5.6 的缓冲池转储与加载 MySQL

对于缓冲池的转储与加载，MySQL 5.6 的 InnoDB 也提供了与 MariaDB 5.5 类似的功能。先

将 InnoDB 缓冲池的页面转储到单独的文件，然后在适当时机将转储的页面加载到 InnoDB 缓冲池。MySQL 5.6 中，对缓冲池的页面转储也并非将 InnoDB 缓冲池的 2~40 GB 数据全部记录下来，所以不用太过担心。缓冲池转储过程中，只是将缓冲池中所有页面所属的表空间与页面 ID 保存到单独的文件。即使缓冲池的大小很大，实际转储也不会花费太多时间。

缓冲池的转储与加载由 MySQL 服务器的系统变量进行控制，下面有 4 个与转储和加载相关的系统变量。MySQL 5.6 中的缓冲池转储或加载与 MariaDB 不同，修改状态变量的值即可转储缓冲池中的信息，或再次将之加载到缓冲池。

innodb_buffer_pool_dump_now

该系统变量常设置为 OFF，若将其设置为 ON，则立即转储 InnoDB 缓冲池中的内容。缓冲池转储结束时，该变量的值立即自动重设为 OFF。

innodb_buffer_pool_load_now

该系统变量的值也常设置为 OFF，将其设置为 ON 时，将读取由 innodb_buffer_pool_filename 系统变量指定的文件，把数据与索引页面加载到 InnoDB 缓冲池。缓冲池加载完毕后，该系统变量重设为 OFF。

innodb_buffer_pool_dump_at_shutdown

若将该系统变量设置为 ON，MySQL 服务器关闭时会将 InnoDB 缓冲池中的内容转储到文件。

innodb_buffer_pool_load_at_startup

若将该系统变量设置为 ON，MySQL 服务器启动时会从 innodb_buffer_pool_filename 系统变量指定的文件读取数据与索引页面，并将其加载到 InnoDB 缓冲池。

看上去只使用 innodb_buffer_pool_dump_at_shutdown 与 innodb_buffer_pool_load_at_startup 两个系统变量就足够了，但其实很多时候更需要使用 innodb_buffer_pool_dump_now 与 innodb_buffer_pool_load_now 两个系统变量。比如，对于不经常使用的大的数据表，假设我们要修改其结构（Schema）。修改表结构之前，先将 InnoDB 缓冲池中的内容进行备份。修改完成后，可以借助 innodb_buffer_pool_dump_now 系统变量再次加载缓冲池。下列示例表示使用 innodb_buffer_pool_dump_now 与 innodb_buffer_pool_load_now 两个系统变量对 InnoDB 缓冲池的内容进行备份再加载的过程。

```
--// 转储缓冲池中的内容
mysql> SET GLOBAL innodb_buffer_pool_dump_now=ON;
Query OK, 0 rows affected (0.06 sec)

--// 查看转储的文件
shell> ls /data/ib_buffer_pool

--// 修改数据表结构
mysql> ALTER TABLE rarely_used_table ADD new_column VARCHAR(20);
```

```
--// 恢复数据表结构修改之前的缓冲池状态
mysql> SET GLOBAL innodb_buffer_pool_load_now=ON;
```

Innodb_buffer_pool_dump_status 与 Innodb_buffer_pool_load_status 是两个与缓冲池转储和加载相关的状态变量，使用它们可以随时了解缓冲池的转储或加载的进展情况。

```
mysql> SHOW GLOBAL STATUS LIKE 'Innodb_buffer_pool_dump_status';
+--------------------------------+-------------+
| Variable_name                  | Value       |
+--------------------------------+-------------+
| Innodb_buffer_pool_dump_status | not started |
+--------------------------------+-------------+

mysql> SHOW STATUS LIKE 'Innodb_buffer_pool_load_status';
+--------------------------------+------------------------+
| Variable_name                  | Value                  |
+--------------------------------+------------------------+
| Innodb_buffer_pool_load_status | Loaded 5121/6441 pages |
+--------------------------------+------------------------+

mysql> SHOW STATUS LIKE 'Innodb_buffer_pool_load_status';
+--------------------------------+----------------------------------------------+
| Variable_name                  | Value                                        |
+--------------------------------+----------------------------------------------+
| Innodb_buffer_pool_load_status | Buffer pool(s) load completed at 140101 02:23:12 |
+--------------------------------+----------------------------------------------+
```

缓冲池转储过程中，即使 InnoDB 的缓冲池很大，也不会花费太多时间，大约 2~3 秒就能完成。当然，进行缓冲池加载时，并非从磁盘逐一读取数据与索引页面，而是以批量方式读取，所以向 InnoDB 缓冲池加载的速度会非常快。但总体而言，缓冲池的加载仍然是项非常耗时的工作。若缓冲池加载的时间非常长，可以将 innodb_buffer_pool_load_abort 系统变量设置为 ON，这样中途即可终止加载处理。

```
mysql> SET GLOBAL innodb_buffer_pool_load_abort=ON;
Query OK, 0 rows affected (0.00 sec)
```

将 InnoDB 缓冲池转储后，删除数据表时，即使存在转储内容中没有的页面，也不会引发什么特殊问题。因此，不必非要在服务器关闭前仅通过转储的文件加载缓冲池。

11. 重做日志大小

为了在服务器崩溃时保证数据安全（Crash-safe）并提升系统性能，InnoDB 存储引擎首先会将提交的事务内容记录到日志文件，而对实际数据文件的修改则放在以后，并以批处理方式

进行处理。我们将这种日志称为重做日志，也称为 WAL（Write Ahead Log，预写式日志）日志，InnoDB 的重做日志像一个逻辑的存储场所，它循环使用多个文件。

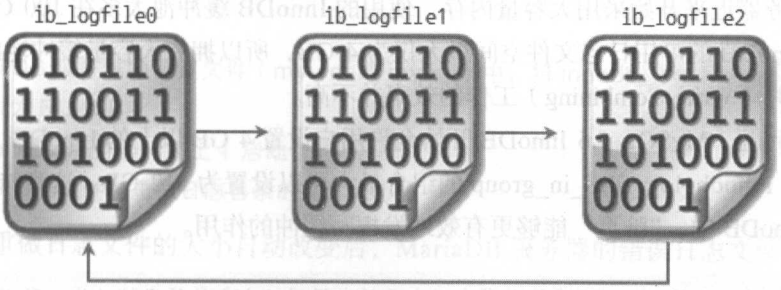


图 6-2 重做日志循环

图 6-2 表示循环重用 InnoDB 日志文件的形态。与 InnoDB 的日志文件相关的 3 个系统变量如下所示。

innodb_log_file_size

该系统变量用于设置各日志文件的大小。MySQL 服务器初始启动时，即按照指定大小创建 InnoDB 日志文件，日志文件的大小不可变。若想改变已经在使用的重做日志文件的大小，必须先彻底关闭 MySQL 服务器，以清除 InnoDB 日志文件的所有内容[①]。

innodb_log_files_in_group

该系统变量用于设置要创建的日志文件的个数。InnoDB 存储引擎最终可使用的日志区域大小为 innodb_log_file_size * innodb_log_files_in_group。

innodb_log_group_home_dir

该系统变量用于指定要创建 InnoDB 重做日志的目录。若不设置该变量的值，则 InnoDB 存储引擎会在 MySQL 服务器的数据目录中创建日志文件。

注意 我们有时会将二进制日志与 InnoDB 的重做日志混淆。有些网站会将 MySQL 服务器的二进制日志与 Oracle 或 MSSQL 的日志进行比较。与 Oracle 或 MSSQL 的日志文件相比，InnoDB 的日志的作用更类似。MySQL 服务器中，二进制日志用于实现 MySQL 服务器独特的复制功能。与 InnoDB 日志不同，二进制日志通常会以非同步的形式被记录到磁盘。

MySQL 5.5 及其以前版本的 InnoDB 可以使用的日志文件最大为 4 GB。InnoDB 日志的整个空间与 InnoDB 缓冲池的“脏页”刷新有着非常密切的关系。InnoDB 缓冲池的大小非常大时，4 GB 大小的日志往往就不够用了。InnoDB 日志空间越大，InnoDB 缓冲池中就可以驻留更多“脏页”。这样，InnoDB 缓冲池的脏页就可以更大批量的形式刷新到磁盘。我们将其称为“写合并”（Write-Combining），更多相关内容将在 6.3.2 节进行详细讲解。

① MySQL 5.5 以前版本中，要想更改重做日志文件的大小，必须先完全关闭服务器，然后删除重做日志文件。修改设置文件并更改好大小后，再重启服务器。但从 MySQL 5.6 开始，只需要在修改设置文件后重启 MySQL 服务器，即可修改重做日志文件的大小。相关内容将在后文进行详细讲解。

“脏页”的变更信息的确会被记录到InnoDB日志文件，但二者的增长速度并不相同。虽然InnoDB“脏页”的增长速度不快，但InnoDB重做日志的文件的使用量可能会急剧增加。此外，越来越多的服务器近来开始采用大容量内存，使用的InnoDB缓冲池大多在100 GB以上。由于MySQL 5.5 InnoDB的可用日志文件空间最大仅为4 GB，所以拥有大容量缓冲池的InnoDB中，很多时候写合并（Write-Combining）工作的效率并不高。

为解决该问题，MySQL 5.6 InnoDB开始允许用户设置4 GB以上的InnoDB日志，innodb_log_file_size * innodb_log_files_in_group的组合最大可以设置为512 GB。这意味着，拥有大容量缓冲池的InnoDB中，“脏页”能够更有效地发挥写缓冲的作用。

提示 MySQL 5.5与5.6的InnoDB存储引擎中，由于并未提供日志文件的存档功能，所以各日志文件的大小并无多大意义。也就是说，拥有1个100 GB日志文件的InnoDB与拥有10个10 GB日志文件的InnoDB在性能与稳定性上并无差异。InnoDB存储引擎中，日志文件由多个文件组成，这并非基于性能的考量，而是操作系统最大文件大小的限制导致的。

InnoDB诞生之初，通过innodb_mirrored_log_groups系统选项可以将InnoDB日志复制到其他位置，然后再进行记录。但目前大量使用的MySQL 5.5或5.6中，该功能已经消失，只在名义上保留着innodb_mirrored_log_groups这一系统变量名。即使将innodb_mirrored_log_groups系统变量设置为2或3，该设置也会被忽略。

12. 更改重做日志大小

MariaDB 5.5或MySQL 5.5中，若想更改InnoDB/XtraDB存储引擎的重做日志文件的大小，整个过程复杂且充满危险。假设MariaDB 5.5中InnoDB/XtraDB的重做日志文件由两个大小为100 MB的文件组成。为了增加重做日志文件的大小，即让该服务器的重做日志文件由2个512 MB的文件组成，整个操作过程如下所示：

(1) 登录MariaDB服务器，执行SET GLOBAL innodb_fast_shutdown=0语句（使InnoDB/XtraDB可以彻底关闭）；
(2) 终止MariaDB服务器；
(3) 打开MariaDB服务器的错误日志文件，检查是否正常关闭（若异常关闭或关闭时发生崩溃，则不可变更重做日志文件的位置）；
(4) 将原重做日志文件备份到另外的目录（使用“mv ib_logfile* /backup_redo/”命令移动到安全目录，原重做日志文件不应该存在于InnoDB/XtraDB重做日志目录）；
(5) 在MariaDB服务器的设置文件（my.cnf或my.ini）中，将innodb_log_file_size变量值从100 MB修改为512 MB；
(6) 启动MariaDB服务器；
(7) 在错误日志与重做日志的目录中查看是否创建了大小为512 MB的重做日志；
(8) 查看是否存在关闭之前的数据；
(9) 完成后，删除备份的重做日志文件（/backup_redo/lib_logfile*）。

对尚不熟悉MySQL服务器或InnoDB/XtraDB的用户而言，上述过程看上去相当复杂，且

操作起来也很有难度。当然，对专家而言，上述操作过程也存在同样风险，也需要注意一些特别的事项。MariaDB 10.0 与 MySQL 5.6 对上述复杂操作过程进行了改善，用户只需完成下列 3 个步骤即可：

(1) 在 MariaDB 服务器的设置文件（my.cnf 或 my.ini）中，将 innodb_log_file_size 变量值从 100 MB 修改为 512 MB；

(2) 重启 MariaDB 服务器（终止 + 启动）；

(3) 查看错误日志文件与重做日志目录的文件大小。

像这样，重做日志文件的大小自动改变后，MariaDB 服务器的错误日志文件中就会记录如下日志。

```
2014-06-23 22:10:57 5048 [Warning] InnoDB: Resizing redo log from 2*3072 to
2*32768 pages, LSN=1626007
2014-06-23 22:10:57 5048 [Warning] InnoDB: Starting to delete and rewrite log files.
2014-06-23 22:10:57 5048 [Note] InnoDB: Setting log file ./ib_logfile101 size to
512 MB
InnoDB: Progress in MB: 100 200 300 400 500
2014-06-23 22:10:58 5048 [Note] InnoDB: Setting log file ./ib_logfile1 size to
512 MB
InnoDB: Progress in MB: 100 200 300 400 500
2014-06-23 22:11:03 5048 [Note] InnoDB: Renaming log file ./ib_logfile101 to ./
ib_logfile0
2014-06-23 22:11:03 5048 [Warning] InnoDB: New log files created, LSN=1626007
```

13. 死锁记录

MySQL 5.5 的 InnoDB 中，即使发生死锁，死锁内容也只能在 SHOW ENGINE INNODB STATUS 命令结果的 DEADLOCK 部分才能看到。更严重的问题是，发生多次死锁时，之前发生的死锁记录会全部删除，使用 SHOW ENGINE INNODB STATUS 命令只能访问最后一次发生死锁的内容。为了解决这一不便，对 MySQL 服务器提供技术支持的公司开发了一些工具，使用它们可以定期连接 MySQL 服务器以访问死锁记录，并将之记录到单独的文件。

MySQL 5.6 的 InnoDB 中，用户不必再使用这些工具记录死锁，它为用户提供了 innodb_print_all_deadlocks 系统变量。发生死锁时，通过 innodb_print_all_deadlocks 系统变量，用户可以选择是否要将死锁的详细内容记录到 MySQL 服务器的错误日志文件。将 innodb_ print_all_deadlocks 系统变量设置为 ON 后，用户可以从 MySQL 服务器的错误日志中查看死锁的详细内容，如下所示。

```
2014-01-25 12:54:16 14dcInnoDB: transactions deadlock detected, dumping
detailed information.
2014-01-25 12:54:16 14dc
*** (1) TRANSACTION:
```

```
TRANSACTION 23566, ACTIVE 22 sec inserting
mysql tables in use 1, locked 1
LOCK WAIT 3 lock struct(s), heap size 320, 2 row lock(s), undo log entries 1
MySQL thread id 4, OS thread handle 0x620, query id 31 localhost 127.0.0.1 root update
insert into test_deadlock values (7)
*** (1) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 71 page no 4 n bits 72 index 'ix_fd' of table 'test'.
'test_deadlock' trx id 23566 lock_mode X locks gap before rec insert
intention waiting
Record lock, heap no 4 PHYSICAL RECORD: n_fields 2; compact format; info bits 0
 0: len 4; hex 8000000a; asc     ;;
 1: len 6; hex 000000195002; asc     P ;;

*** (2) TRANSACTION:
TRANSACTION 23565, ACTIVE 30 sec inserting, thread declared inside InnoDB 5000
mysql tables in use 1, locked 1
3 lock struct(s), heap size 320, 2 row lock(s), undo log entries 1
MySQL thread id 3, OS thread handle 0x14dc, query id 32 localhost 127.0.0.1 root update
insert into test_deadlock values (7)
*** (2) HOLDS THE LOCK(S):
RECORD LOCKS space id 71 page no 4 n bits 72 index 'ix_fd' of table 'test'.
'test_deadlock' trx id 23565 lock_mode X locks gap before rec
Record lock, heap no 4 PHYSICAL RECORD: n_fields 2; compact format; info bits 0
 0: len 4; hex 8000000a; asc     ;;
 1: len 6; hex 000000195002; asc     P ;;

*** (2) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 71 page no 4 n bits 72 index 'ix_fd' of table 'test'.
'test_deadlock' trx id 23565 lock_mode X locks gap before rec insert intention
waiting
Record lock, heap no 4 PHYSICAL RECORD: n_fields 2; compact format; info bits 0
 0: len 4; hex 8000000a; asc     ;;
 1: len 6; hex 000000195002; asc     P ;;
*** WE ROLL BACK TRANSACTION (2)
```

6.3.2 “脏页”刷新

在InnoDB中处理用户查询后，其结果内存空间中的缓冲池数据已经发生变化，但尚未将其记录到磁盘。我们将这种页面（块）称为“脏页”，将“脏页”记录到磁盘上称为“刷新”。InnoDB的“脏页”刷新大致可分为Flush-list刷新与LRU刷新两类。InnoDB缓冲池中的页面主要使用LRU_list与Flush_list两个链表进行管理，LRU_list用于管理缓冲池中非常用页面的列表，Flush_list依据时间顺序管理缓冲池中更改页面的列表。LRU_list与Flush_list都能引发“脏

页”刷新。

下面介绍 InnoDB 存储引擎的核心功能，如下所示。

- InnoDB 的重做日志与缓冲池“脏页”间的关系
- DBMS 的 WAL 机制
- MySQL 5.5 InnoDB 与 MariaDB 5.5 XtraDB 的“脏页”刷新算法
- MySQL 5.6 InnoDB 与 MariaDB 10.0 XtraDB 的“脏页”刷新算法

此外，为了更好地学习这些内容，还将向各位介绍一些必知术语。

注意 本部分提及的 InnoDB 是 XtraDB 与 InnoDB 的统称，而提到的 XtraDB 则是指除 InnoDB 之外的 XtraDB 存储引擎本身。由于 InnoDB 与 XtraDB 存储引擎的基本操作方式相同，故将之统称为 InnoDB。需要区分 InnoDB 与 XtraDB 时，将分别使用 MySQL InnoDB 与 XtraDB 指称，请不要混淆。

1. Flush_list 刷新

MySQL 5.5 服务器的 InnoDB 中，对于“脏页”刷新还存在许多不足之处。学习 MySQL 与 MariaDB 的“脏页”刷新之前，先了解 InnoDB 的重做日志文件与缓冲池的“脏页”之间的关系。

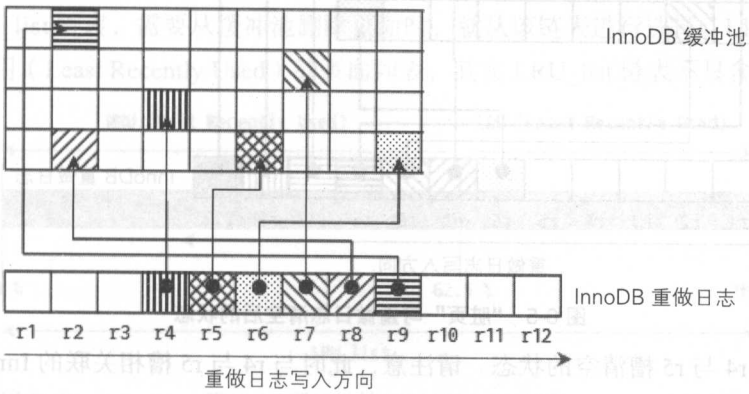


图 6-3 InnoDB 缓冲池与重做日志间的关系

图 6-3 表示 InnoDB 存储引擎中重做日志与缓冲池“脏页”之间的关系。InnoDB 的重做日志由多个文件组成，但 InnoDB 存储引擎在内部会将其全部聚拢，识别为一个连续空间，所以图 6-3 使用一个连续空间表示。图 6-3 中，假设重做日志的使用顺序为 r1 到 r12。那么使用 r4~r9 槽后，重做日志的各个槽借助箭头连接到 InnoDB 缓冲池中的各页面。此处箭头表示根据相应重做日志槽进行修改的 InnoDB 缓冲池的“脏页”。

在图 6-2 的基础上对数据略作修改，再使用 3 个重做日志槽，如图 6-4 所示。

在图 6-3 中再执行 3 个 UPDATE 语句，将在重做日志中再使用 r10~r12 槽，图 6-4 表示该状态下 InnoDB 缓冲池与重做日志的状态。图 6-4 中，r4 槽是目前 InnoDB 存储引擎中最旧的重做日志槽，r12 槽存储着最近更改的内容。图 6-4 中可以看到多用的 3 个重做日志槽，InnoDB 存储引擎在适当的时候应该清空旧的重做日志槽。

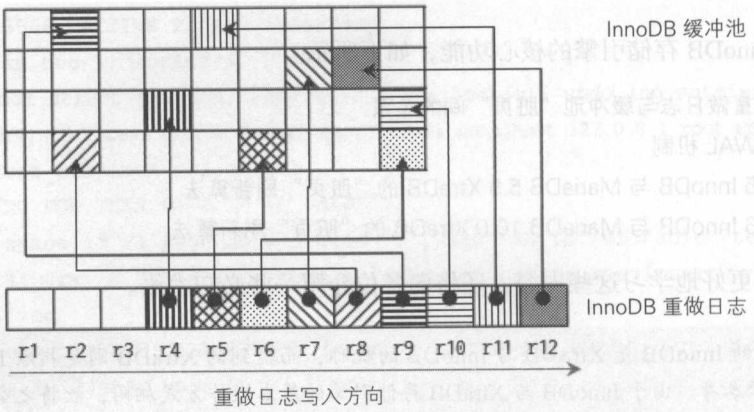


图 6-4 发生更多写入后重做日志与缓冲池的关系

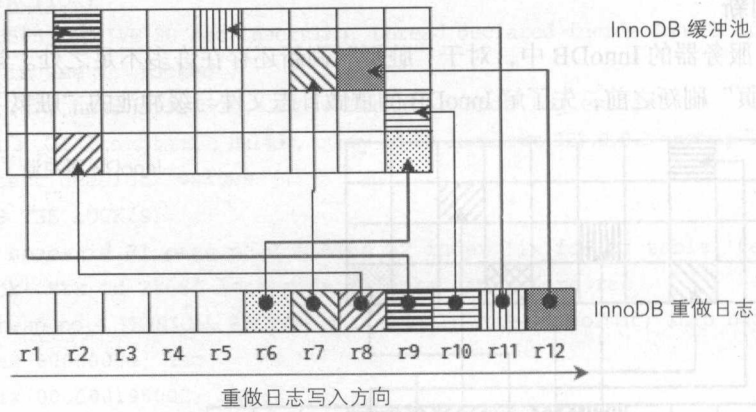


图 6-5 “脏页”与重做日志清空后的状态

图 6-5 表示 r4 与 r5 槽清空的状态。请注意，此时与 r4 与 r5 槽相关联的 InnoDB 缓冲池的“脏页”消失了。根据 RDBMS 重做日志的工作方式（Write Ahead Log），在重做日志与“脏页”之中，重做日志总是要首先被刷新到磁盘。并且只有 InnoDB 缓冲池中的“脏页”完全刷新到磁盘时，才能再次使用与该“脏页”相关的重做日志槽。此时，为了在缓冲池中快速查找“脏页”，会根据“脏页”的修改时间顺序为其维护一个链表，该链表称为 Flush_list。

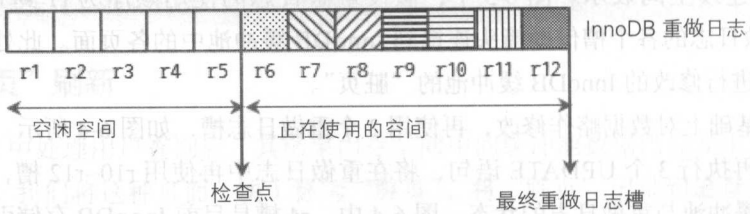


图 6-6 重做日志

图 6-6 只展现了图 6-5 中的重做日志，在当前状态将重做日志分为空闲空间与正在使用的空

间两部分。处于使用状态的空间中，最旧的重做日志槽是最近检查点出现的地点。图 6-6 中，从“（最近的）检查点”到“最终重做日志槽”这段长度，InnoDB 将其称为检查点年龄（Checkpoint age）。

此处有意思的是，重做日志必须保证有许多最大空闲空间，InnoDB 缓冲池的“脏页”维持的时间越久，越有效率。一个最好的方法时，保留适当的重做日志空闲空间，同时最大限度维持缓冲池的“脏页”。但 MySQL 5.5 的 InnoDB 存储引擎中，处理这部分时无法保持它们之间的均衡关系。MariaDB 的 XtraDB 也存在这种不均衡的问题，为此添加了许多算法进行改善。虽然这些改善并不完美，但比 MySQL 的 InnoDB 有很大进步。

2. LRU_list 刷新

除了 Flush_list 外，InnoDB 还管理着另外一种链表，即 LRU_list 链表。例如，假设 InnoDB 的缓冲池全部由索引与数据页填满，即 InnoDB 缓冲池中没有空闲空间。此情形下，若一个查询请求到来，为了执行该查询，要从磁盘再读 2~3 个页面，这样会发生什么呢？由于缓冲池中再无空闲空间，InnoDB 存储引擎必须从缓冲池中删除目前已经加载到其中的几个页面，以创建空闲空间。此时，InnoDB 存储引擎并不会随意从缓冲池中选择要删除的页面。InnoDB 存储引擎管理着 LRU_list 链表，需要从缓冲池删除页面时，就从该链表进行选择。LRU_list 链表维持着最近最不常用（Least Recently Used）的页面列表，其实 LRU_list 链表不只含有不常用页面。

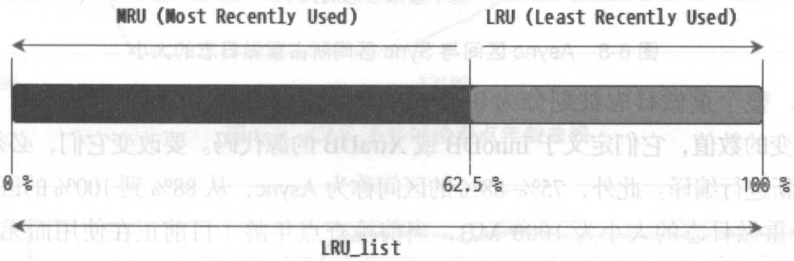


图 6-7 缓冲池的 MRU 与 LRU 链表

图 6-7 表示 InnoDB 的 LRU_list 结构体。由图可知，LRU_list 分为两个区域。这两个区域的分割点为 62.5%（整个链表的 5/8 位置），MRU（Most Recently Used）区域位于整个 LRU_list 的 5/8 之前，其余的部分为 LRU（Least Recently Used）区域。也就是说，5/8 分割点之前的区域存放着经常使用的页面，其余 3/8 区域存放着不怎么使用的页面。修改 innodb_old_blocks_pct 系统变量的值后，可以更改 MRU 与 LRU 之间的分割点（默认值为 5/8）。

需要从缓冲池中选择几个页面进行删除时，InnoDB 存储引擎会引用 LRU_list。在最不常用的 LRU 链表的末端选择几个页面，成为缓冲池中要删除的“牺牲者”（Victim）。假如 InnoDB 从 LRU_list 中选择了最后的 3 个页面进行删除。如果要删除的 3 个页面中有 2 个“脏页”，会怎么样呢？此情形下，必须先将“脏页”记录（刷新）到磁盘，然后再删除。

像这样，与 InnoDB 的重做日志完全无关，为了向缓冲池写入新页面，有时也需要将页面刷新到磁盘，这称为 LRU_flush。本章凡是提到“脏页刷新”都是指 Flush_list 刷新（Flushlist_

flush），LRU_list 刷新单独表述为“LRU 刷新”。MySQL 5.5 之前，Flush_list 刷新大部分都有优化代码，而 LRU 刷新部分几乎不带优化功能。因此，MySQL 5.5 针对 LRU 刷新引起的性能低下问题完全没有提供相应的调优选项。直到 MySQL 5.6 才引入针对 LRU 刷新的调优选项，有关内容将在后文进行讲解。

3. InnoDB 与 XtraDB 的“脏页”刷新

由于 XtraDB 存储引擎是在 InnoDB 代码的基础上改进而成的，所以 XtraDB“脏页”刷新的基本工作方式与 InnoDB 一样。如图 6-6 所示，InnoDB 与 XtraDB 中，从最后一个检查点出现时所在重做日志的位置（LSN，Log Sequence Number）到最后重做日志槽所在位置，位于两端之间的这段长度称为“检查点年龄”。“检查点年龄”之所以重要是因为，要使用 InnoDB 与 XtraDB 全部重做日志空间的大小与检查点年龄的长度确定要进行刷新的“脏页”数量。根据检查点年龄的长度，可能阻止 XtraDB 或 InnoDB 的部分查询或全部查询。

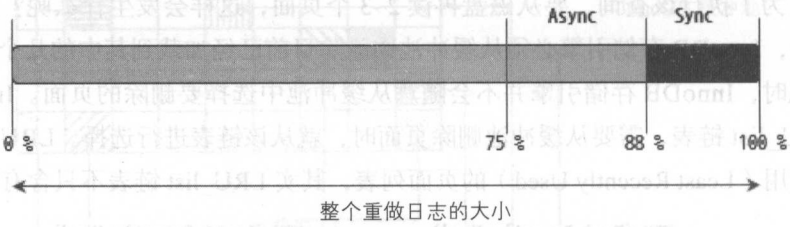


图 6-8 Async 区间与 Sync 区间所占重做日志的大小

图 6-8 中，整个重做日志被划分为 0~75%、75%~88%、88%~100% 三个区间。其中，75% 与 88% 是固定不变的数值，它们定义于 InnoDB 或 XtraDB 的源代码。要改变它们，必须修改 MySQL 源代码，并重新进行编译。此外，75%~88% 的区间称为 Async，从 88% 到 100% 的区间称为 Sync。

假设整个重做日志的大小为 1000 MB，当前检查点年龄（目前正在使用而无法重新使用的重做日志区间）为 800 MB（75%~88% 的区间）左右。此时，InnoDB 与 XtraDB 工作模式与平时略有不同，我们将之称为 Async 模式。若情形更一步恶化，检查点年龄增长到 900 MB（88%~100% 区间）左右，此时的工作模式与平时完全不同，称为 Sync 模式。因此，各区间的名称分别为 Async 与 Sync。那么日常模式、Async 模式、Sync 模式有何不同呢？

日常模式

由于重做日志的空闲空间足够多，所以 InnoDB 或 XtraDB 会使用指定算法将“脏页”适当刷新到磁盘。检查点年龄低于 75% 时，进入 MySQL 服务器的查询执行将不受任何限制。

Async 模式

重做日志的空闲空间仍有一定闲余，但正在逼近不足状态。该状态下会组织执行到来的查询。InnoDB 或 XtraDB 为了确保重做日志的空闲空间会阻止来自客户的查询，并将缓冲池中的“脏页”刷新到磁盘。当然，已经处于执行状态的查询会继续执行而不会被阻止。也就是说，Async 模式下会组织一部分查询，执行其余查询。

Sync 模式

重做日志的空闲空间处于极度缺乏的状态，该状态下，当前来自客户端的查询以及已经在执行的查询都会被阻止，并且会非常积极地将缓冲池中的“脏页”刷新到磁盘。也就是说，Sync 模式下，进入 XtraDB 或 InnoDB 的所有查询都会被阻止执行，并且积极地将“脏页”刷新到磁盘。

InnoDB 或 XtraDB 进入 Async 或 Sync 模式时，缓冲池中的“脏页”会以最快速度刷新到磁盘，以重新使用重做日志。也就是说，XtraDB 或 InnoDB 进入 Async 或 Sync 模式时，来自应用程序的查询会发生延迟，MySQL 服务器也开始显现异常。

稳定运行的 MySQL 服务器中，检查点年龄不会进入 Async 或 Sync 模式，而始终处于整个重做日志大小的 75% 之内，显示稳定曲线。图 6-9 与图 6-10 表示检查点年龄的一定模式，表明运行稳定。

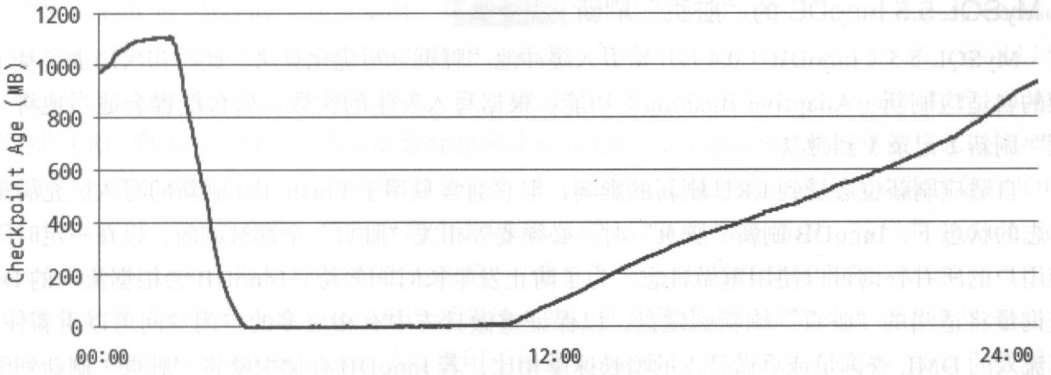


图 6-9 DML 不多时检查点年龄曲线

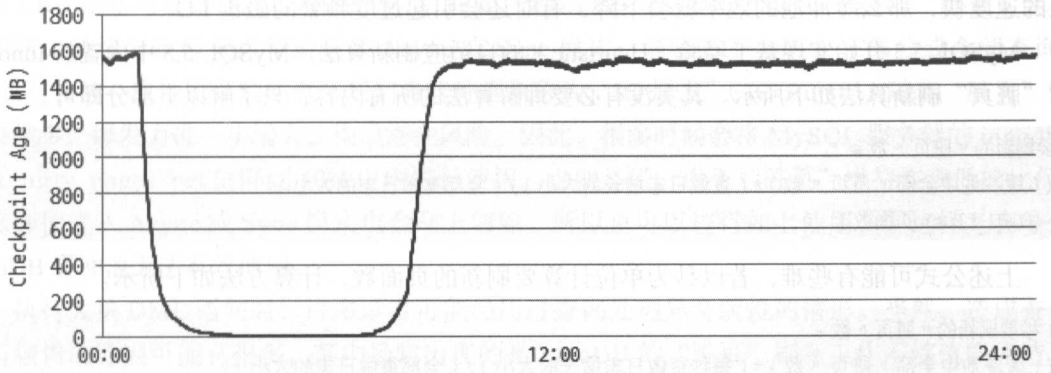


图 6-10 DML 很多时检查点年龄曲线

根据服务的 DML 请求程度，图 6-9 与图 6-10 都呈现一定模式，只要不形成锯齿形状的曲线，就表明服务相当稳定。与图 6-9 相比，图 6-10 中的曲线不受用户查询流入量太大的影响，绘制的检查点年龄都在一定级别。该状态下，重做日志的空闲空间得到一定保证，InnoDB 的缓冲池也保持一定“脏页”，处于一种非常稳定的状态。

但实际使用MySQL服务器提供的服务中，用户查询的类型都各不相同。此外，由于重做日志与“脏页”的生成量并非总以相同比率增长，且每时每刻流入的查询量也不同，所以图6-10中的检查点年龄曲线其实是很难看到的。虽然我们很难创建最优状态，但至少能分辨XtraDB或InnoDB当前是否在最优状态下工作。若各位正管理着MySQL服务器，还没有监测过InnoDB或XtraDB的检查点年龄变化的情形，建议大家现在就查看检查点年龄曲线。

根据检查点年龄的不同，InnoDB或XtraDB有3种工作模式。若想让MySQL服务器工作在稳定模式下，必须保持适当的检查点年龄，使服务器维持日常模式，而不要进入Sync或Async模式。好的“脏页”刷新算法并不在于如何结束Async与Sync模式并返回日常模式，而在于如何防止进入Async与Sync模式并在日常模式下进行“脏页”刷新。与Oracle的InnoDB相比，XtraDB拥有的优点之一是进一步改善了算法，以决定在日常模式下要将多少“脏页”刷新到磁盘。

4. MySQL 5.5 InnoDB的“脏页”刷新 MySQL

MySQL 5.5（InnoDB 1.0.4）开始引入缓冲池“脏页”写优化算法。此时初次添加了基于经验的自适应刷新（Adaptive flushing）功能。根据写入发生的次数，后台线程会适当地将“脏页”刷新（记录）到磁盘。

自适应刷新也会受到LRU刷新的影响，但它通常只用于Flush_list刷新的写入。充满重做日志的状态下，InnoDB刷新“脏页”时，必须要等相关“脏页”全部被刷新，以在一定时间内使用户的所有查询可以使用重做日志。为了防止发生长时间等待，InnoDB会根据流入的DML查询量将适当的“脏页”刷新到磁盘，以保证重做日志中有相应多的空闲空间可以重新使用。与流入的DML查询量或重做日志的增长速度相比，若InnoDB存储引擎将“脏页”刷新到磁盘的速度比较慢，那么就会阻止用户查询。反之，若将“脏页”刷新到磁盘的速度比重做日志增长的速度快，那么缓冲池的效率就会下降，有时还会引起过度频繁的磁盘I/O。

MySQL 5.5开始实现基于经验（Heuristic）的自适应刷新算法。MySQL 5.5中内置的InnoDB的“脏页”刷新算法如下所示。其实没有必要理解算法的所有内容，只了解以下部分即可。

```
要刷新的"脏页"数=
((缓冲池中全部"脏页"数)*(重做日志增长的大小)/(全部重做日志的大小))
-(LRU_flush页面数)
```

上述公式可能有些难，若以秒为单位计算要刷新的页面数，计算方法如下所示。

```
1秒要刷新的"脏页"数=
((缓冲池中全部"脏页"数)*(每秒重做日志增长的大小)/(全部重做日志的大小))
-(每秒刷新的LRU_flush页面数)
```

其实，MySQL 5.5的自适应刷新在应用时并不像其名称那样“柔韧”。InnoDB的检查点年龄比较低时，它工作在非Async或Sync模式的日常模式下。将根据上述公式的计算结果，把适当的“脏页”刷新到磁盘。但重做日志累积的速度比流入的DML查询（INSERT、UPDATE、DELETE）量更快时，检查点年龄就会增高，InnoDB可能会进入Async或Sync模式。

为了应对自适应刷新无法消化服务流量的情形，MySQL 5.5 提供了两个用于调优的系统变量。

innodb_io_capacity

通过上述公式计算要刷新的“脏页”数，若计算结果大于 innodb_io_capacity 系统变量的设置值，则只会按 innodb_io_capacity 设置的数量刷新“脏页”。也就是说，innodb_io_capacity 变量用于设置 1 次可刷新的最大“脏页”数。

innodb_max_dirty_pages_pct

该系统变量用于设置 InnoDB 缓冲池中可以保持的最大“脏页”数，其值是一个百分比。默认值为 75，表示缓冲池中允许的最大“脏页”数可以达到总页数的 75%。MySQL 5.5 的 InnoDB 中，若“脏页”所占比例超过了 innodb_max_dirty_pages_pct 变量所设的值，自适应刷新算法就会停止，以 innodb_io_capacity 变量所设值的 2 倍速度刷新“脏页”。

设置 innodb_io_capacity 系统变量时，只要将其设为所用硬盘的 IOPS（Input Output Operation Per Second）即可。MySQL 5.5 的 InnoDB 中，若缓冲池中“脏页”所占比例很高，则可以将 innodb_io_capacity 变量值适当设置得大一些，以降低“脏页”所占比例。但是，由于 InnoDB 内部会将 LRU 刷新与重做日志的写次数合起来计算 innodb_io_capacity，所以重做日志以同步模式工作的 MySQL 服务器中，有时需要将 innodb_io_capacity 变量的值设置得比实际磁盘的 IOPS 更大。从以往的实际经验看，装配有两个 SAS 磁盘的 MySQL 服务器中，曾经将 innodb_io_capacity 设置为 1500。即使设置如此大的值，“脏页”所占比例也没有下降。MySQL 5.5 的 InnoDB 并未将“检查点年龄”这一概念暴露给用户，所以管理者很难对其进行监视并调优。

innodb_max_dirty_pages_pct 系统变量用于设置 InnoDB 缓冲池允许的最大“脏页”数所占总页数的比例值。但如前所述，MySQL 5.5 的 InnoDB 中的“脏页”刷新算法还不够成熟，又受到 LRU 刷新与重做日志写入的影响，“脏页”无法以适当速度进行刷新。这样，“脏页”会不断积压，最终导致检查点年龄进入 Async 或 Sync 模式。该状态下会阻止一部分或全部查询，“脏页”会被一次性刷新到磁盘，我们将这种现象称为“IO 爆发”（IO Burst）。随着“脏页”数越来越多，爆发力进一步增大，构成潜在风险。因此，很多时候会将 MySQL 服务器的 innodb_max_dirty_pages_pct 值降到 10% 以内进行使用。也就是说，由于“脏页”维持在很低的比例，那么即使进入 Async 或 Sync 模式也会马上解除，所以也可以进行如上使用。但这样又会导致 InnoDB 缓冲池无法有效使用。

执行大量 DML 语句时，许多读者可能经历过查询处理异常缓慢的情形。当然，造成查询执行缓慢的原因可能有很多，其中经常出现的是 InnoDB 的“脏页”刷新工作不够流畅而造成的延迟。PerconaServer 与 MariaDB 5.5 内置的 XtraDB 中，为了解决这一问题，使用多种方法对自适应刷新进行了大量改进。

5. MariaDB 5.5 XtraDB 的“脏页”刷新 MariaDB

MariaDB 5.5 使用内嵌 XtraDB 取代 MySQL 的 InnoDB 存储引擎。MariaDB 即使要创建使用 InnoDB 存储引擎的数据表，其内部生成的其实是使用 XtraDB 存储引擎的数据表。XtraDB

在InnoDB基础上做了大量改进，但最具代表性的内容是对“脏页”刷新算法的改善。XtraDB存储引擎新增了innodb_adaptive_flushing_method、innodb_flush_neighbor_pages、innodb_checkpoint_age_target三个系统变量，以对“脏页”刷新进行更精细的调整。

MySQL 5.5的InnoDB存储引擎中，检查点年龄超过全部重做日志大小的75%时，转入Async模式；超过88%时，进入Sync模式。也就是说，工作模式由全部重做日志的大小自动确定。但根据流入MySQL服务器的DML的特点，75%与88%这两个基准值有时会发生变化。MySQL 5.5的InnoDB中，需要根据流入的DML的特点修改重做日志的大小。其实，修改重做日志大小时，不仅要重启MySQL服务器，重启时还必须“干净关闭”（Clean shutdown），使重做日志的内容可以不进行恢复。XtraDB中，可以通过innodb_checkpoint_age_target系统变量调整转入Async与Sync模式的检查点年龄的比例。也就是说，使用innodb_checkpoint_age_target系统变量能够改变整个重做日志的大小，获取增减效果。innodb_checkpoint_age_target是动态变量，更改时即使不重启MariaDB服务器，也能根据服务特点对重做日志的大小逐步进行调整。

(1) innodb_adaptive_flushing_method

MySQL 5.5的InnoDB只能控制是否要使用自适应刷新功能，而XtraDB对此进行了进一步改善，通过innodb_adaptive_flushing_method系统变量可以选择使用哪种自适应刷新算法。在innodb_adaptive_flushing_method系统设置中可以设置native、estimate、keep_average。

native

即使使用XtraDB存储引擎，“脏页”刷新算法也采用与MySQL 5.5的InnoDB算法相同的自适应刷新。换言之，开启该选项将不使用XtraDB中添加的“脏页”刷新优化。

estimate

检查点年龄没有达到innodb_checkpoint_age_target系统变量设置值的1/4时，每10秒就会刷新innodb_io_capacity变量所设数量的“脏页”。检查点年龄超过innodb_checkpoint_age_target的1/4时，将以秒为单位，依据重做日志增加的比例将“脏页”刷新到磁盘。当然，检查点年龄超过1/4点时，仍然会以10秒为单位继续刷新innodb_io_capacity所设数量的“脏页”。

keep_average

自适应刷新设置为keep_average算法时，将以0.1秒为单位进行刷新“脏页”。keep_average算法也会根据重做日志的增长比例适当地将“脏页”刷新到磁盘。由于以0.1秒为单位的重做日志的增长量会反映在算法中，所以将“脏页”刷新到磁盘时会快速反映流入的DML的变化。此外，keep_average算法中以0.1秒为单位发生“脏页”刷新，并可以维持一致的磁盘I/O。keep_average模式是针对FushionIO或FlashMax等SSD存储器设计的。

(2) innodb_flush_neighbor_pages

MySQL 5.5的InnoDB刷新“脏页”时，并非将其逐页刷新到磁盘。若逐页写入磁盘，每个页面都要采用随机I/O进行。在有磁盘旋转的HDD中，这种工作方式相当没有效率。因此，InnoDB会将要刷新的“脏页”周围的“脏页”一并收集起来，将其一次性刷入磁盘。MySQL

5.5 的 InnoDB 存储引擎无法修改该算法，但 XtraDB 借助 innodb_flush_neighbor_pages 系统变量可以调整为 3 种模式。

area

将 innodb_flush_neighbor_pages 系统变量设置为 area 后，将采用与 MySQL 5.5 相同的“脏页”刷新方式。该模式下，以要刷新的“脏页”为基准，从相邻的 128 个页面中收集所有“脏页”并刷新。图 6-11 中，刷新 D0 时将从左右相邻的 128 个页面中选择处于“脏页”状态的 D1~D9 页面一起刷新。

cont

将 innodb_flush_neighbor_pages 系统变量设置为 cont 后，只将要刷新的“脏页”及其相邻的“脏页”（紧密相邻且处于“脏页”状态的页面）一起收集并刷新。图 6-11 中，以 D0 为基准，只刷新 D3 与 D4、D5 与 D6 页面。

none

将 innodb_flush_neighbor_pages 系统变量设置为 none 后，只刷新目标“脏页”，完全不考虑相邻“脏页”。area 或 cont 刷新模式面向随机 I/O 性能较差的 HDD 硬盘，是批量刷新“脏页”的方式。SSD（Solid State Drive）提供了非常高的随机 I/O 能力，此时再花费额外成本查找并刷新“脏页”反而会降低效率。因此，FusionIO 或 FlashMax 这类随机 I/O 处理速度非常快的系统中，最好将 innodb_flush_neighbor_pages 变量设置为 none。设置为 none 后，图 6-11 中要刷新的对象就只有 D0 页面了。

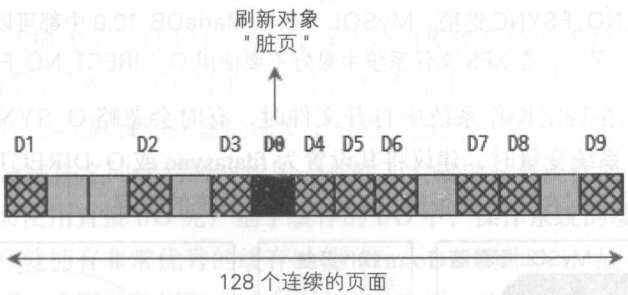


图 6-11 刷新相邻的“脏页”

(3) innodb_flush_method

XtraDB 的刷新功能中，另一个改善内容是针对直接 I/O（Direct IO）的。MySQL 5.5 的 InnoDB 中，通过 innodb_flush_method 系统变量设置可以决定读写数据文件的方式。MySQL 5.5 可以将 innodb_flush_method 系统变量设置为如下三个值之一。

fdatasync

fdatasync 刷新方法用于与之前版本保持兼容。但是，其实不能在 MySQL 服务器的设置文件中像 innodb_flush_method=fdatasync 这样进行设置。若想设置为 fdatasync，则需要将 innodb_flush_method 设置本身从 MySQL 服务器的设置文件中删除。fdatasync 原来用于通过 fdatasync() 系统调用保持数据文件的写同步，但现在用于在 MySQL 内部实现对 fsync() 的调用，不再使用 fdatasync() 系统调用。fdatasync() 系统调用原来只对文件数据进行同步，而不会对文件的元数据（比如文件的访

问时间、文件大小等）进行同步。但常用的 Linux 中，fdatasync() 系统调用与 fsync() 得到相同处理。这样，fdatasync 原来的目的就不复存在了。

O_DSYNC

写数据文件时，InnoDB 存储引擎会先利用 O_DSYNC 选项打开数据文件，写入完成后，再通过 fsync() 系统调用进行同步。O_DSYNC 虽然利用 O_SYNC 选项打开日志文件，但并不进行刷新操作。根据操作系统的种类不同，打开文件时，O_SYNC 选项有时会被忽略。因此，将 innodb_flush_method 系统变量设置为 O_DSYNC 可能不是好的选择。

O_DIRECT

写数据文件时，InnoDB 存储引擎先利用 O_DIRECT 选项打开数据文件，写入完成后，再通过 fsync() 系统调用进行同步。

ALL_O_DIRECT

MariaDB 5.5 的 XtraDB 中新增的刷新方法，讲解 XtraDB 内容时会再次说明[①]。

O_DIRECT_NO_FSYNC

MySQL 5.6 开始引入的刷新方法。原来 O_DIRECT 文件输入输出模式在写操作时，内部会调用 fsync()，O_DIRECT_NO_FSYNC 拥有与之相同的效果。但 MySQL 的 InnoDB 存储引擎中，在 O_DIRECT 模式下进行写操作后也会额外再调用 1 次 fsync()。为了避免这种重复操作，MySQL 5.6 引入了 O_DIRECT_NO_FSYNC 选项。MySQL 5.6 与 MariaDB 10.0 中都可以使用 O_DIRECT_NO_FSYNC 刷新方式。另外，在 XFS 文件系统中最好不要使用 O_DIRECT_NO_FSYNC[②]。

前文已经提到，在特定操作系统中打开文件时，有时会忽略 O_SYNC 选项。因此，设置 innodb_flush_method 系统变量时，建议将其设置为 fdatasync 或 O_DIRECT 两者之一。fdatasync 与 O_DIRECT 最大的不同在于，是否使用 Linux 操作系统的缓存。

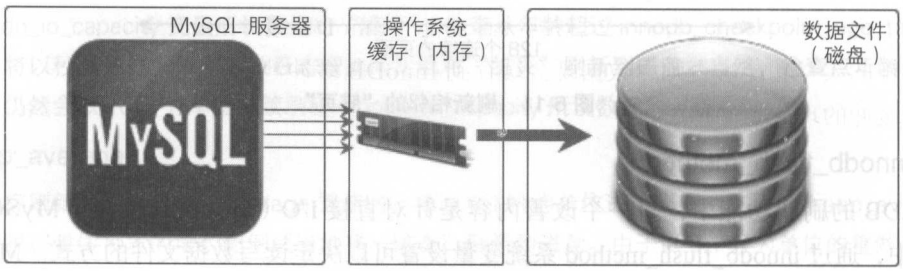


图 6-12 Cached I/O

图 6-12 表示基于 Linux 缓冲的（Cached）I/O，将 innodb_flush_method 系统变量设置为 fdatasync 或 O_DSYNC 模式时，采用此种操作方式。Linux 的所有磁盘读写操作通常都基于

① MySQL 并不支持 ALL_O_DIRECT 选项，只有 MariaDB 支持该刷新方式，但此处为了保持说明的一致性而将其一起包含。

② O_DIRECT_NO_FSYNC 选项是从 MySQL 5.6 开始支持的刷新方式，但此处为了保持说明的一致性而将其一起包含。

Cached I/O。基于缓存的 I/O 模式下，读写磁盘数据操作过程如下所示。

从磁盘读数据

基于缓存的 I/O 中，应用程序读取磁盘数据时，会首先查找 Linux 操作系统的缓存，查看所需磁盘块数据是否已经缓存。若缓存中存在要查找的数据，则直接将缓存数据返回给应用程序；否则就会实际读取磁盘获取需要的数据，将之先复制到操作系统的缓存，然后返回给应用程序。以后，应用程序再次请求相同数据块时，不需再读磁盘，直接从缓存将数据返回给应用程序即可，这大大加快了处理速度。

向磁盘写数据

应用程序向操作系统请求写文件数据时，Linux 接受请求后只将其复制到操作系统的缓存。然后，Linux 在某个合适的时间点将从应用程序收集到的所有写请求一次性写入磁盘。

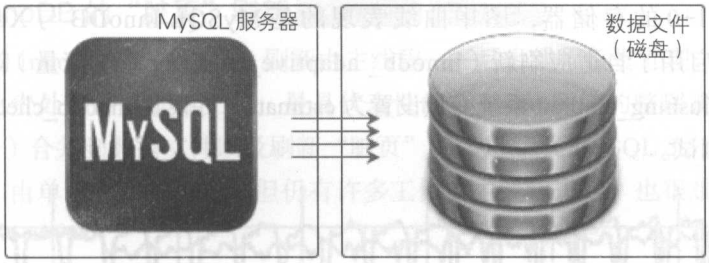


图 6-13 直接 I/O

图 6-13 以图示方式描述了直接 I/O。直接 I/O 不经过操作系统的缓存，与图 6-12 中的基于缓存的 I/O 相比，其读写过程变得更简单。由于读写磁盘不经过另外的模块，所以相当直观。与 CPU 或内存性能相比，实际读写磁盘的速度十分慢。因此，基于缓存的 I/O 中，操作系统的缓存在快速硬件与慢速硬件之间起到缓冲作用，有助于大幅提升系统性能。既然如此，为何不使用基于缓存的 I/O 而使用直接 I/O 呢？基于缓存的 I/O 中，操作系统的缓存区域是系统内存。然而由于 InnoDB 本身就拥有非常优秀的缓存功能（InnoDB 缓冲池），故不需要再浪费系统主存提供这样的缓存功能。否则，数据既会缓存到 InnoDB 缓冲池，也会缓存到操作系统的缓存，产生双重缓冲的现象。这种双重缓冲会造成内存产生不必要的浪费。

有 InnoDB 缓冲池的前提下，即使采用直接 I/O 方式读磁盘，性能也不会下降多少。但与基于缓存的 I/O 相比，直接 I/O 的写性能的确会低一些。近来，磁盘的 RAID 控制器都额外装有 512 MB 到 1~2 GB 的缓存内存，所以经常采用直接 I/O 方式进行写磁盘而不会引起太大问题。当然，若 RAID 控制器中装有 BBU（Battery-Backed Unit，电池备份单元），则可以进一步保证数据的稳定性，更锦上添花。也许有人认为使用直接 I/O 会提升性能，再强调一下，使用直接 I/O 功能绝不是为了提升性能，而是为了防止双重缓冲，从而更有效地使用内存。

此外，与基于缓存的 I/O 相比，Linux 之所以偏爱使用直接 I/O 是由 Linux 缓存管理模块特点造成的。Linux 缓存模块会尽力缓存尽可能多的数据，也尽力保持更多的缓存数据。因此，有时会因 Linux 的缓存模块所使用的内存而造成 InnoDB 的缓冲池内存经常被意外换出（Swap-out）到磁盘。为了防止出现这样的问题，最好在读写 Linux 数据文件时强制不使用操作系统的

缓存，所以使用直接 I/O 是最简单快捷的方法。

再回到 innodb_flush_method 系统变量。将 innodb_flush_method 设置为 O_DIRECT 后，数据文件将不再使用操作系统的缓存，但 InnoDB 的重做日志仍然会使用基于缓存的 I/O。XtraDB 添加了 ALL_O_DIRECT 选项，将 innodb_flush_method 系统变量设置为该选项后，连重做日志也可以在不使用操作系统缓存的情形下被记录到磁盘。使用该选项后，InnoDB 的数据文件和重做日志均可使用直接 I/O。

(4) XtraDB 存储引擎的改进效果

有很多 INSERT 查询时，在 MySQL 5.5 的 InnoDB 中，所增重做日志的相应“脏页”无法适当刷新，InnoDB 存储引擎会定期进入 Async 模式。图 6-14 中，MySQL 服务器使用 SAS 磁盘组成的 RAID 1+0 的存储器，图中曲线表现的是 MySQL InnoDB 与 XtraDB 的处理量。MySQL InnoDB 启用了自适应刷新（innodb_ adaptive_flushing 系统变量）功能，XtraDB 将 innodb_adaptive_flushing_method 系统变量设置为 estimate，并设置 innodb_checkpoint_age_target 的状态下所做的测试。

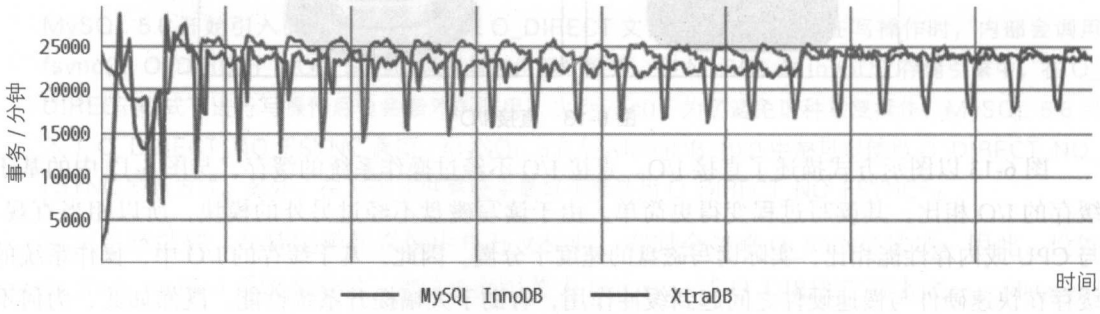


图 6-14 RAID 10（SAS 磁盘）（estimate）

图 6-15 是对使用 PCI-e 类型的 SSD 存储器的 MySQL InnoDB 与 XtraDB 存储引擎的测试结果。当然，InnoDB 开启了自适应刷新功能，XtraDB 将 innodb_adaptive_flushing_method 系统变量设置为 keep_average。

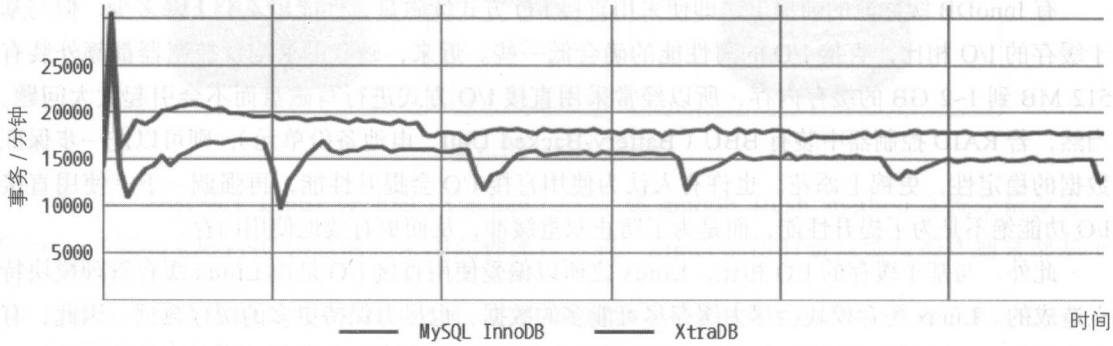


图 6-15 SSD（PCI-e）（keep_average）[①]

① 图 6-14 与图 6-15 中呈现锯齿形状的曲线为 MySQL InnoDB，表现平缓性能的曲线为 XtraDB。

从图 6-15 中可以看到，在 XtraDB 存储引擎中，RAID 1+0 组成的 HDD 与 SSD 的吞吐量都保持稳定。但 MySQL 5.5 的 InnoDB 存储引擎中，吞吐量定期表现出急剧下降又急剧上升的现象。图中曲线并没有表现出 InnoDB 的监测点年龄的变化。但 MySQL 5.5 的 InnoDB 存储引擎中，吞吐量急剧下降时，监测点年龄就会增加。InnoDB 存储引擎先进入 Async 模式，然后再转入正常状态。像这样，InnoDB 存储引擎进入 Async 模式后就会刷新缓冲池中的“脏页”，并且阻止或无法快速处理用户请求。

提示 图中曲线引用了 Percona Performance Blog 的性能基准测试结果，原文参见下列 URL。
http://www.mysqlperformanceblog.com/2010/12/20/mysql-5-5-8-and-percona-server-being-adaptive/

6. MySQL 5.6 InnoDB 的“脏页”刷新 MariaDB MySQL

MySQL 5.5 的 InnoDB 中，“脏页”刷新由主线程——后台线程负责处理，除了“脏页”刷新外，主线程还负责处理许多其他工作，最具代表性的有 Undo 区域的整理删除、将改变缓冲区（Change buffer）合并到数据文件以及刷新“脏页”等。虽然从 MySQL 5.5 开始，Undo 区域的整理与删除工作由单独的线程负责，但仍有许多工作由主线程进行，也很难避免它们之间相互干涉。

因此，MySQL 5.6 开始引入了页面清理者（Page Cleaner）线程，专门负责“脏页”刷新。MySQL 5.6 InnoDB 的页面清理者线程执行“脏页”刷新工作时，遵循以下两项标准。

访问类型（Access Pattern）

缓冲池无空闲空间时，就会依据 LRU_list 将最近最少使用的页面刷新到磁盘。要刷新的“脏页”顺序由 LRU_list 管理。

年龄（Age）

Flush_list 链表会依据变更的时间顺序对“脏页”进行管理，并使用基于几种经验的算法刷新 Flush_list 中的页面。

(1) 访问类型

MySQL 5.5 InnoDB 中，与访问类型相关的 LRU_list 刷新并非由后台线程负责，而由用户线程处理。也就是说，用户的查询请求到来时，为了处理该请求，InnoDB 存储引擎有时必须读取磁盘上的“脏页”。此时，若缓冲池的 Free_list 中无空闲页面，则 LRU_list 中最后的页面就会刷新到磁盘。但更大的问题是，刷新 LRU_list 中的页面时并不只刷新一两页，而是以批量方式将很多页面刷到磁盘。像这样，Free_list 中无空闲页面时，执行查询可能需要很多时间。因此，MySQL 5.6 InnoDB 对此进行了进一步改善，让后台线程处理 LRU_list 的刷新工作。

MySQL 5.6 InnoDB 针对 LRU_list 刷新引入了名为 innodb_lru_scan_depth 的新系统变量。该系统变量可以针对每个缓冲池实例分别进行设置，查找 page cleaner 线程要刷新的“脏页”时，借助该变量可以设定要在 LRU_list 中扫描的页数。后台线程每秒执行 1 次 LRU 刷新时，在 IO 比较空闲的 MySQL 服务器中，可以将 innodb_lru_scan_depth 设置得大一些；而在写入操作多且

缓冲池大的 MySQL 中，需要适当地将其减小。

(2) 检查点年龄

InnoDB 缓冲池中可保留的“脏页”数由重做日志大小的总和确定，同样，Flush_list 中可保留的页面总量也由 InnoDB 中所有重做日志大小的总和决定。反之，为了在 InnoDB 的重做日志文件中留下适当大小的空闲空间，就要不断刷新 Flush_list 中的“脏页”。但如果对 Flush_list 中的“脏页”刷新过多，就会导致缓冲池的效率下降；而如果刷得太慢，又很难保证 InnoDB 重做日志有足够的空闲空间。当然，如果 InnoDB 重做日志中的空闲空间即将耗尽，MySQL 服务器的所有查询都会进入阻止状态，InnoDB 存储引擎会“拼命地”将 Flush_list 中的“脏页”刷新到磁盘。为了防止出现这一情况，在保证 InnoDB 重做日志有空闲空间的同时，又要让缓冲池保留适当的“脏页”，并定期进行管理。

对于要刷新到磁盘的页面数，MySQL 5.6 的 InnoDB 会以当前检查点年龄为基础，使用下列公式计算。

```
要刷新的脏页数 =
((innodb_io_capacity_max/innodb_io_capacity)*
(lsn_age_factor*sqrt(lsn_age _factor)))/7.5;
```

上述计算公式中，lsn_age_factor 是由最大可允许检查点年龄与当前检查点年龄的比例计算的值。由上述公式可知，MySQL 5.6 的 InnoDB“脏页”的刷新速度不仅由 innodb_io_capacity 系统变量决定，也由新增的 innodb_io_capacity_max 系统变量决定。

MySQL 5.6 InnoDB 提供了 innodb_io_capacity 与 innodb_io_capacity_max 两个系统变量，以使后台线程可以控制“脏页”刷新。下面介绍如何使用这两个系统变量以及如何调优。

innodb_io_capacity

与缓冲池的“脏页”或改变缓冲（Change buffer）的索引合并一样，需要将许多这样的数据记录到磁盘，这些工作全部由 InnoDB 的后台线程负责处理。InnoDB 的后台线程处理这些操作并写磁盘时，通过 innodb_io_capacity 系统变量可以限制所允许的最大写磁盘的次数。

该系统变量的默认值为 200，在可以承受高负荷或磁盘性能很好的系统中，可以适当地将其值设置得大一些。由于该设置变量是动态变量，所以 MySQL 服务器处于运行状态下也可以修改其值。innodb_io_capacity 与缓冲池的实例个数无关，用于控制所有缓冲池的全部写磁盘的次数。若该值设置得太大，缓冲池的脏数据就会快速刷新到磁盘，从而降低缓冲池的效率；若其值设置得太小，缓冲池的全部“脏页”数就会增加，重做日志的空闲空间就会减少。

最近常用的 15 000 RMP 或 10 000 RMP 的磁盘中，设置变量值时，每个磁头算作 200~300 即可。例如，在一个由 4 个 10 000 RMP 磁盘组成的 RAID 1+0 系统中，用于写磁盘的磁头有 2 个，那么只要将 innodb_io_capacity 设置为 400~600 左右即可。

innodb_io_capacity_max

与 InnoDB 重做日志的增长量相比，若“脏页”刷新延迟，InnoDB 就会更“激进地”执行刷新工作，可能会超过 innodb_io_capacity 指定的 IOPS 值。这种危急情形下，innodb_io_capacity_max 系统变

量用于设置所允许的最大 IOPS，这样，即使发生 IO 爆发（Burst），也能保证磁盘 IO 不会耗尽所有系统资源。

该系统变量的设置方法与设置 innodb_io_capacity 变量的方法非常相似，只要设置一个比 innodb_io_capacity 变量值更大的值即可。innodb_io_capacity_max 的默认值为 innodb_io_capacity 默认值的两倍，直接设置时最少要设置 2000 以上的值。innodb_io_capacity _max 变量的设置值并不针对单个缓冲池实例，而是指所有缓冲池实例中发生的磁盘 I/O 总和。

此外，MySQL 5.6 InnoDB 存储引擎还新引入了 innodb_adaptive_flushing_lwm、innodb_max_dirty_pages_pct_lwn、innodb_flushing_avg_loops 系统变量。这些选项用作自适应刷新（innodb_adaptive_flushing=ON 时）算法的参数，用于更精细地调整自适应刷新的工作方式。MySQL 5.5 的 InnoDB 中已经存在的 innodb_adaptive_flushing、innodb_io_capacity、innodb_max_dirty_pages_pct 选项采用与以前相同的方式，但其功能由 MySQL 5.6 的 InnoDB 新增的 3 个选项进行限制或扩展。

innodb_adaptive_flushing_lwm

该选项充当重做日志文件大小的“低潮线”（low water mark），若重做日志文件的使用量超过 innodb_adaptive_flushing_lwm 设置值，则即使未激活 innodb_adaptive_flushing，也会自动启动自适应刷新。

innodb_max_dirty_pages_pct_lwn

InnoDB 不允许缓冲池中存在超过 innodb_max_dirty_pages_pct 变量所设比例的“脏页”。innodb_max_dirty_pages_pct 的默认值为 75%，为了降低“脏页”比例，innodb_max_dirty_pages_pct_lwn 系统变量充当执行额外“脏页”刷新（Pre-Flushing）的“低潮线”。innodb_max_dirty_pages_pct_lwn 系统变量的默认值为 0，即额外“脏页”刷新默认处于关闭状态。

innodb_flushing_avg_loops

该系统变量用于决定是否继续维持以前刷新状态的计算值以及维持次数（Iteration）。随着流入的 DML 查询的变化，该系统变量设置会对 InnoDB 存储引擎的“脏页”刷新速度产生影响。

将 innodb_flushing_avg_loops 系统变量设置为较大的值时，InnoDB 会更长久地使用以前计算的值，让自动适应刷新反应更慢一些。将 innodb_flushing_avg_loops 系统变量设置得较高时，用户查询的流入速度将不会立即对后台的“脏页”刷新产生影响。MySQL 服务器中，只有重做日志的使用量低于 75% 且缓冲池的“脏页”比例维持在低于 innodb_max_dirty_pages_pct 变量设置的水平时，才建议各位将 innodb_flushing_avg_loops 变量值设置得高一些。

流入 MySQL 服务器的查询请求均衡且重做日志文件大小较大时，建议将 innodb_flushing_avg_loops 系统变量设置得大一些。其值设置得越大，MySQL 服务器工作得就会更温和一些。流入 MySQL 服务器的查询量的变化严重且重做日志剩余的空闲空间又不多时，建议将 innodb_flushing_avg_loops 系统变量设置得小一些。其值越小，“脏页”刷新就能越快地对 DML 查询请求做出反应，这样就能将重做日志的使用量维持在比较合适的水平。

innodb_flush_neighbors

该系统设置与 MariaDB 5.5 XtraDB 中使用的 innodb_flush_neighbor_pages 系统设置具有相同功能。

各位若忘记了，可参考图 6-11。MySQL 5.6 的 InnoDB 中，对于属于一个区间（Extent）[①] 的连续页面，该选项用于决定是否收集相邻“脏页”并采用批量方式刷新。将该系统变量设置为 0 时，将不对相邻“脏页”使用批量刷新功能；设置为 1 时，只对同一区间中的连续“脏页”进行批量刷新；设置为 2 时，相关“脏页”所属区间中的所有“脏页”会一起刷新到磁盘。

7. MariaDB 10.0 的 XtraDB MariaDB

MySQL 5.6 的 InnoDB 存储引擎中，刷新“脏页”时所用算法的计算公式如下所示。

```
要刷新的脏页数 =
 ((innodb_io_capacity_max/innodb_io_capacity) *
  (lsn_age_factor * sqrt(lsn_age _factor)))/7.5;
```

由以上公式可知，innodb_io_capacity_max 系统变量的设置值对计算实际要刷新到磁盘的“脏页”数有很大影响。合并“改变缓冲”或服务器处于无 DML 查询流入的禁用状态、正在关闭状态时，innodb_io_capacity 系统变量决定“脏页”刷新的工作量。

也就是说，服务器处于积极处理 DML 查询的状态时，要调整 innodb_io_capacity_max 系统变量的值。服务器处于禁用状态或正在关闭状态时，要调整 innodb_io_capacity 值决定“脏页”刷新的工作量。此外，改变缓冲（change_buffer 系统变量）设置激活时，innodb_io_capacity 所设值的 5% 通常用于执行插入缓冲合并；而插入缓冲使用量超过 50% 时，innodb_io_capacity 的 5%~55% 将用于进行插入缓冲合并。激活“改变缓冲”而使服务器处于无 DML 查询的禁用状态时，将根据 innodb_io_capacity 变量设置的 IOPS 执行插入缓冲合并。

XtraDB 存储引擎采用了新算法，与 MySQL 5.6 内置的刷新算法相比，新算法通过多种分析使缓冲区更有效率，同时保证重做日志的空闲空间。XtraDB 使用如下公式计算要刷新的“脏页”数。

```
要刷新的脏页数 =
 (((srv_max_io_capacity/srv_io_capacity) *
  (lsn_age_factor* lsn_age_factor * sqrt(lsn_age _factor)))/700.5))
```

其实，MySQL 5.6 的 InnoDB 使用的公式与 XtraDB 中使用的公式看上去几乎没什么差别。但很多时候，这些细微的差别形成的优化类型会产生很大差异。

图 6-16 中的曲线表示“脏页”刷新速度，由 MySQL 5.6 的 InnoDB 存储引擎与 XtraDB 的检查点年龄变化决定。图中两条曲线最大的差别是，各检查点年龄地点的倾斜度。理想的曲线是，在低检查点年龄时间点上，“脏页”刷新速度低，缓冲池的写缓冲与批量写（Write Combining）拥有较高的工作效率。检查点年龄增大时，“脏页”刷新速度就会增加，重做日志的空闲空间产生得就快。从这个意义上看，比较图 6-16 中的两条曲线就会发现，XtraDB 的前进方向更加理想。

---

① 每个区间的页面数随页面大小不同而不同，页面大小为 16 KB 时，一个区间有 64 个页面；页面大小为 8 KB 时，一个区间含有 124 个页面。

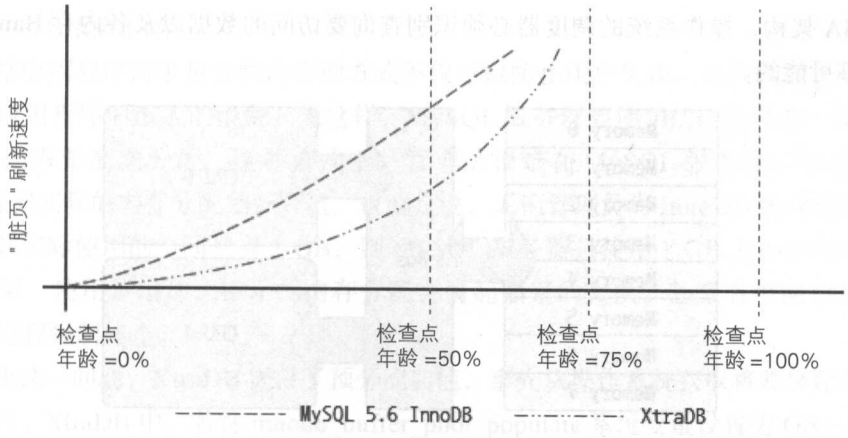


图 6-16 MySQL 5.6 InnoDB 与 MariaDB 10.0 XtraDB 的“脏页”刷新类型

注意 图 6-16 引用了 Percona Performance Blog 的图表，原文参见下列 URL。
http://www.mysqlperformanceblog.com/2013/10/30/innodb-adaptive-flushing-in-mysql-5-6-checkpoint-age-and-io-capacity/

6.3.3 缓冲池性能改善 MariaDB

InnoDB 存储引擎最核心的部分是缓冲池。缓冲池与 InnoDB 的大部分功能有关，也是 InnoDB 所有代码块中使用最频繁的模块。因此，缓冲池与性能有着非常密切的联系。XtraDB 存储引擎改善了许多与缓冲池相关的功能，下面逐一介绍 XtraDB 中独有的功能。请注意，这些功能在 MySQL InnoDB 中是不支持的。

1. NUMA

AMD 的 Opteron 与英特尔的 Nehalem 处理器开始采用 NUMA（Non-Uniform Memory Access，非一致内存访问）架构。在 NUMA 架构之前，处理器采用的是 UMA 架构。首先，简单了解有关 UMA 与 NUMA 架构的结构。

图 6-17 为 UMA 架构。UMA 架构中，CPU 0 与 CPU 1 全部连接到系统总线（Bus），内存被整合在一起，访问内存必须且只能通过系统总线进行。这种架构中，CPU 访存路径长，且花费的时间也比较长。

图 6-18 为新的 NUMA 架构。此架构下，内存直接与 CPU 连接在一起。图 6-17 与图 6-18 中，CPU 是实际的物理 CPU 芯片，并不是通常所说的物理或逻辑内核。NUMA 架构中，CPU 0 运行应用程序时，将使用 0~3 号内存。从图中可以看到，由于内存与 CPU 直接相连，所以 CPU 访问内存的速度相当快。

令人遗憾的是，以上介绍的 NUMA 架构的优点不适用于 MySQL 服务器这类 RDBMS。原因在于，MySQL 服务器的 InnoDB 存储引擎起初会在 1 个线程中将非常大的内存分配为缓冲池，处理查询时也几乎不存在“只读取特定内存区域即可”的情形。其实，RDBMS 若要正常

使用NUMA架构，操作系统的调度器必须识别查询要访问的数据以及各内存Bank中的数据，这几乎是不可能的。

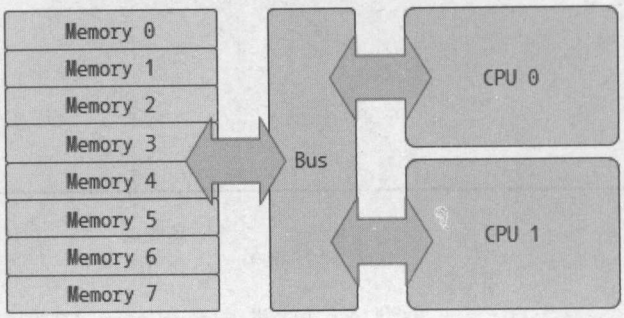


图 6-17 UMA 架构

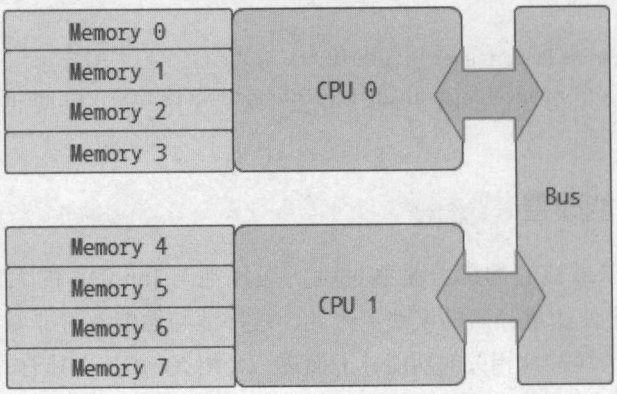


图 6-18 NUMA 架构

MySQL服务器中，NUMA架构会引起令人头疼的交换（swap）问题。一个拥有2个CPU且采用NUMA架构的系统中，各CPU可用内存分别减半。假如其中一个CPU用掉了90%以上的内存，即使另一个CPU可用内存的50%处于空闲状态，也会引发交换问题。因此，在Linux中启动MySQL服务器时默认关闭NUMA，或者使用numactl实用工具让内存返回轮循方式（Round-Robin）并进行分配。XtraDB提供了numa_interleave系统变量，通过该系统变量，MariaDB服务器能够以Interleaving模式（以轮询方式分配内存）启动。

当然，使用numactl实用工具让MySQL服务器以Interleaving模式进行工作并非难事。只要直接修改mysqld_safe或my.cnf文件，即可让MySQL或MariaDB以Interleaving模式工作。

2. 缓冲池内存初始分配

在Linux中启动MySQL服务器时，很难一开始就获得所需的全部内存。当然，可以通过修改Linux的默认内存分配方式实现，但由于要改变Linux的默认工作方式，所以也不能轻松实现。Linux将InnoDB缓冲池设置为40 GB，启动MySQL服务器后，InnoDB存储引擎也不是从一开始就占用40 GB内存，只是向Linux预约40 GB内存。只有应用程序进行访问时，Linux

才向应用程序分配预约内存。

这种根据应用程序需求量分配内存的方式不仅可以防止用户失误，也可以防止将不必要的内存分配给应用程序而造成的浪费。像这样，MySQL 服务器要使用的内存并非一次性分配完毕，而是根据需要渐次分配。这种方式下，管理员很难在 MySQL 服务器启动的同时把握 MySQL 服务器实际的内存分配是否合适。也就是说，无论管理员为 InnoDB 缓冲池分配再多内存空间，若其实际使用的空间只有 1 GB，则 MySQL 服务器只使用 1 GB 大小的内存。但以后随着数据积累、使用量增加，错误的内存分配会引起频繁的交换，甚至有可能会被 Linux 的 OOM Killer 进程强制终止。

为了解决这一问题，XtraDB 提供了预分配功能，事先从操作系统获取将要分配给缓冲池的全部内存空间。XtraDB 中，若将 innodb_buffer_pool_populate 系统变量设置为 ON，在 MySQL 服务器启动的同时，XtraDB 申请为缓冲池分配内存时会得到所设置的全部 40 GB 的内存。并且，由于以后执行查询时，即使数据量增多也不会再为缓冲池分配内存空间，所以全部内存的使用量是固定的。这样，管理员在 MySQL 服务器启动的同时就会知道针对 MySQL 服务器的内存设置是否存在问题。

3. InnoDB 的锁细分

MySQL 5.5 的 InnoDB 中存在着臭名昭著的 kernel_mutex 互斥体。InnoDB 中，很难创建另外互斥体的代码块内几乎都使用 kernel_mutex。因此，进行完全无关的处理时，经常会因 kernel_mutex 导致不必要的等待。MariaDB 5.5 的 XtraDB 将 kernel_mutex 进一步细分，大大减少了锁竞争。与 XtraDB 类似，MySQL 5.6 InnoDB 也进行相似的改进，对 kernel_mutex 进一步细分，以减少锁竞争。

此外，MySQL 5.6 InnoDB 中，从 LRU_list 删除页面时，为了掌握页面数量而使用了 buffer_pool 互斥体。XtraDB 去除了使用互斥体，虽然准确性有些下降，但实际访问缓冲池的瓶颈却得到大大改善。即使对 LRU_list 页面数把握的准确程度有所下降，但对 InnoDB 存储引擎的整体处理没有任何影响。

MySQL 5.6 InnoDB 为了保持缓冲池中各链表的同步，将一个全局使用的互斥体细分为如下几个互斥体，以降低缓冲池的锁竞争。

flush_state_mutex

保护刷新状态的“脏页”的互斥体

LRU_list_mutex

缓冲池中对存储在 LRU 链表的页面进行同步的互斥体

flush_list_mutex

缓冲池中对要刷新的“脏页”链表进行同步的互斥体

free_list_mutex

缓冲池中管理未使用空闲页链表的互斥体

zip_free_mutex

为了处理压缩页面而管理所用空闲页列表的互斥体

zip_hash_mutex

保护检索压缩页面的散列表的互斥体

像 MySQL 5.5 所做的那样，Percona 一直在对 MySQL 5.6 进行技术支持。发现这种不必要的锁竞争后，XtraDB 对此做出相应改善，并继续推出新的改良版本，防止类似的锁竞争对同步性产生损害。

4. 基于 I/O 的负载性能提升

磁盘数据文件的大小要比 InnoDB 缓冲池大得多，MySQL 服务器中的读磁盘操作极其频繁时，经常会发生下列这些现象。

- 页面清理线程（Page cleaner thread）中，与准备的可用页相比，缓冲池的 Free_list 中的可用页消耗得更快。
- 缓冲池的 Free_list 中的可用页消耗殆尽，查询或其他后台线程一直处于等待状态。
- 用于同步 Free_list 的互斥体等待增多。
- 页面清理线程与用户线程中，即使准备有可用页面，但由于放入缓冲池的 Free_list 时要等待互斥体，这样会降低页面清理线程的性能。

为了阻止上述恶性循环，XtraDB 为生成可用页（Free page）的页面清理线程赋予优先级，使之能够获得可用页链表的互斥体。当然，即使为可用页生成者（页面清理线程）赋予优先级，也不能完全消除互斥体的竞争。即使 Free_list 中没有任何可用页，Free_list 互斥体仍会被先占用再释放，并以此循环。Free_list 中无可用页时，页面清理线程与用户线程会同时准备可用页，并将其添加到 Free_list。用户线程每次准备 1 个可用页，这会引起对 Free_list 互斥体的更多竞争。

因此，Free_list 中无可用页时，XtraDB 的用户线程不会执行 LRU_list 刷新，而是调整 innodb_empty_free_list_algorithm 系统变量，使页面清理线程以批量方式执行。设置 innodb_empty_free_list_algorithm 系统变量时，可以从 legacy 与 backoff 中选择其一。若设置为 legacy，则采用 MySQL 5.6 原有算法；若设置为 backoff，则使用 XtraDB 的新算法。

5. 自适应散列分区

InnoDB 存储引擎中，频繁对主键或索引加锁时，就会进行优化。即针对相应索引的部分值，在内部自动生成散列索引，从而加快搜索的执行速度。此时，内部自动生成的索引称为“自适应散列索引”。对于 MySQL 5.6 的 InnoDB 自适应散列索引，在多核系统中大量执行读写混用的查询时，会引发同步问题。

为了解决自适应散列索引的同步问题，XtraDB 引入了 innodb_adaptive_hash_index_partition 系统变量。XtraDB 会根据 innodb_adaptive_hash_index_partition 系统变量设置的整数值创建相应数量的自适应散列索引，并以 InnoDB 数据表的各索引 ID 为基准，确定要使用的自适应散列索

引。也就是说，每个数据表只有 1 个索引的 MySQL 服务器中，自适应散列索引分区可能没有多大意义。若将 innodb_adaptive_hash_index_partition 系统变量设为 1，则操作方式与 MySQL 5.6 的 InnoDB 相同；设为大于 2 的值时，则根据设定的个数创建自适应散列索引。innodb_adaptive_hash_index_partition 变量的取值范围为 1~64（此时为 64 位平台，32 位平台下的最大值为 32）。

6.3.4 原子写（针对 FusionIO SSD 的 Atomic write） MariaDB

为了防止重做日志空间浪费，InnoDB 存储引擎的重做日志只记录页面修改的内容。因此，InnoDB 存储引擎刷新“脏页”到磁盘文件时，若发生问题而只记录了一部分，那么相应页面的内容可能无法恢复。像这样，页面只记录一部分的现象称为局部页面（Partial-page 或 Torn page）。硬盘运转不正常或系统异常终止时，都会发生这样的问题。

为了防止出现该问题，InnoDB 存储引擎使用了 Double-write 技术。图 6-19 表示 InnoDB 的 Double-write 的工作原理。如图 6-19 所示，假设 InnoDB 要将“脏页”A~F 刷新到磁盘。此时，InnoDB 存储引擎将修改的内容实际记录到磁盘文件前，先将“脏页”A~F 聚拢，通过一次磁盘写操作，将其写到系统表空间的 DoubleWrite 缓冲区。之后，InnoDB 存储引擎将各“脏页”逐个随机写到文件的适当位置。

“脏页”A~F 正常写入实际的数据文件时，就不再需要记录在系统表空间的 DoubleWrite 缓冲区中的更改内容。DoubleWrite 缓冲中的内容原本只是为了应对写实际数据文件的过程中发生的失败。假设页面 A 与 B 均正常写入，而操作系统在页面 C 写入过程中异常终止。重启时，InnoDB 存储引擎总是将 DoubleWrite 缓冲中的内容与数据文件的页面全部进行比较，若存在内容不同的页面，则将 DoubleWrite 缓冲的内容复制到数据文件的页面。通过 innodb_doublewrite 系统变量可以控制是否使用 DoubleWrite 功能。

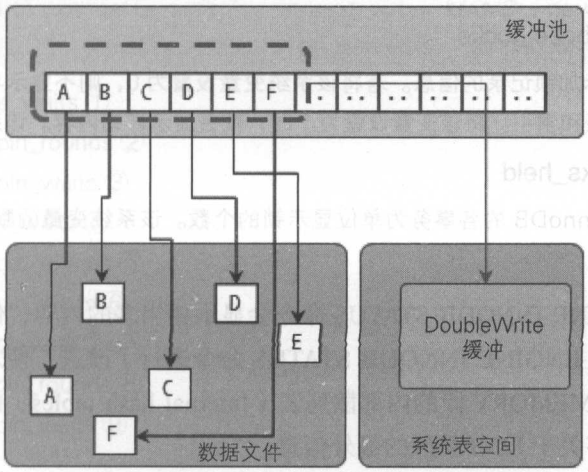


图 6-19 Double Write 工作方式

DoubleWrite 缓冲常用于保证数据安全，类似 HDD 这种有磁盘旋转的存储系统中，由于要进行 1 次顺序写磁盘，故不会有什么问题。但像 SSD 这类随机 I/O 与顺序 I/O 所花成本差不多的存储系统中，却是相当大的负担。SSD 存储系统中，DoubleWrite 缓冲最终刷新两个“脏页”。FusionIO 这类高性能的 SSD 拥有以原子为单位处理写磁盘的功能。此处所说的“以原子为单位”是指，InnoDB 存储引擎向磁盘写入 16 KB 数据时，FusionIO 驱动器会采用事务方式进行处理。也就是说，或者将 16 KB 完全写入磁盘，或者一点也不写入磁盘。

灵活使用 FusionIO 的原子写功能后，就不再需要使用 InnoDB 存储引擎的 DoubleWrite 写功能。也就是说，可以将 SSD 磁盘的写入量减少一半。XtraDB 将 innodb_use_atomic_write 系统变量设置为 ON，即可以使用 FusionIO 的原子写功能。将 innodb_use_atomic_write 设置为 ON 后，若 innodb_flush_method 为 fdatasync 或 O_DSYNC，则自动更改为 O_DIRECT。innodb_flush_method 为 O_DIRECT、ALL_O_DIRECT、O_DIRECT_NO_FSYNC 三者之一时，将保持不变。innodb_use_atomic_write 为 ON 时，innodb_doublewrite 会自动设置为 OFF。

注意 FusionIO SSD 中，只有在使用 DirectFS 的情形下才可能将 innodb_use_atomic_write 系统变量设置为 ON。FusionIO 之外的其他 SSD 产品仍然不支持 Atomic Write 功能，XtraDB 必须使用 FusionIO 的 Atomic Write API。

此外，即使使用 FusionIO SSD，若不使用 DirectFS 而使用 EXT3、XFS、EXT4 文件系统，也同样无法使用 Atomic Write 功能。

6.3.5 扩展的 InnoDB 引擎状态输出 MariaDB

MySQL 服务器中，使用 SHOW ENGINE INNODB STATUS 命令显示的内容相当重要。显示的内容太过准确，导致内容可读性下降。XtraDB 引入了 2 个系统变量，用于控制如何显示 SHOW ENGINE INNODB STATUS 命令的执行结果。

innodb_show_verbose_locks

该变量决定是否显示加锁记录的信息。若将该系统变量设置为 0，则不显示与锁相关的详细内容，只显示锁了哪些数据表和索引。将该变量设置为 1 时，就会显示详细内容。该系统变量的默认值为 0。

innodb_show_locks_held

该变量决定是否以 InnoDB 的各事务为单位显示锁的个数。该系统变量的默认值为 10，最大可设为 1000。

使用 SHOW ENGINE INNODB STATUS 命令会显示相当多的内容，但仍部分所需信息未显示。XtraDB 对 SHOW ENGINE INNODB STATUS 命令进行了改进，使之可以字节为单位显示 BUFFER POOL AND MEMORY 段的内部散列表（Internal hash tables）的大小与缓冲池的全部大小，还可以显示更多关于 LOG 节区的额外信息。

为了监测 MySQL 服务器的 InnoDB 引擎，使用 SHOW ENGINE INNODB STATUS 命令显示的结果中，要查看许多值。但由于 MySQL 5.6 的 InnoDB 中，SHOW ENGINE INNODB

STATUS 命令的结果是以文本形式显示的，所以只有分析所有值才能监测相应状态值。但是，SHOW ENGINE INNODB STATUS 命令输出的结果非常长，从结果中提取所需值并非易事。XtraDB 从 SHOW ENGINE INNODB STATUS 命令执行的结果中将主要监视对象的状态值抽取为 MariaDB 状态变量，以便于访问。也就是说，不必分析 SHOW ENGINE INNODB STATUS 命令的执行结果，而使用 SHOW GLOBAL STATUS 命令监测 XtraDB 引擎的状态值。

1. 与后台线程有关的状态变量

XtraDB 将 SHOW ENGINE INNODB STATUS 命令的执行结果中有关 BACKGROUND THREAD 的状态值注册为 3 个状态变量。

```
BACKGROUND THREAD
-----------------
srv_master_thread loops: 1 srv_active①, 0 srv_shutdown, 11844 srv_idle②
srv_master_thread log flush and writes: 11844③
```

- Innodb_master_thread_active_loops ①
- Innodb_master_thread_idle_loops ②
- Innodb_background_log_sync ③

2. 与信号量相关的状态变量

XtraDB 将 SHOW ENGINE INNODB STATUS 命令的执行结果中有关 SEMAPHORE 的状态值注册为 9 个新的状态变量。

```
SEMAPHORES
----------
OS WAIT ARRAY INFO: reservation count 9664, signal count 11182
Mutex spin waits 20599③, rounds 223821②, OS waits 4479①
RW-shared spins 5155⑤, OS waits 1678④; RW-excl spins 5632⑧, OS waits 2592⑦
Spin rounds per wait: 10.87 mutex, 15.01 RW-shared⑥, 27.19 RW-excl⑨
```

- Innodb_mutex_os_waits ①
- Innodb_mutex_spin_rounds ②
- Innodb_mutex_spin_waits ③
- Innodb_s_lock_os_waits ④
- Innodb_s_lock_spin_rounds ⑤
- Innodb_s_lock_spin_waits ⑥
- Innodb_x_lock_os_waits ⑦
- Innodb_x_lock_spin_rounds ⑧
- Innodb_x_lock_spin_waits ⑨

3. 与插入缓冲及自适应散列索引相关的状态变量

XtraDB 将 SHOW ENGINE INNODB STATUS 命令的执行结果中有关 INSERT BUFFER

AND ADAPTIVEHASH INDEX 的状态值注册为 10 个新的状态变量。

```
INSERT BUFFER AND ADAPTIVE HASH INDEX
-------------------------------------
Ibuf: size 1⑩, free list len 0④, seg size 2⑨, 0 merges⑧
merged operations:
 insert 0⑦, delete mark 0⑤, delete 0⑥
discarded operations:
 insert 0③, delete mark 0①, delete 0②
Hash table size 174431, node heap has 0 buffer(s)
0.00 hash searches/s, 0.00 non-hash searches/s
```

- Innodb_ibuf_discarded_delete_marks ①
- Innodb_ibuf_discarded_deletes ②
- Innodb_ibuf_discarded_inserts ③
- Innodb_ibuf_free_list ④
- Innodb_ibuf_merged_delete_marks ⑤
- Innodb_ibuf_merged_deletes ⑥
- Innodb_ibuf_merged_inserts ⑦
- Innodb_ibuf_merges ⑧
- Innodb_ibuf_segment_size ⑨
- Innodb_ibuf_size ⑩

4. 与日志相关的状态变量

XtraDB 将 SHOW ENGINE INNODB STATUS 命令的执行结果中有关 LOG 的状态值注册为 5 个新的状态变量。

```
LOG
---
Log sequence number 10145937666①
Log flushed up to   10145937666②
Pages flushed up to 10145937666
Last checkpoint at  10145937666③
Max checkpoint age    80826164⑤
Checkpoint age target 78300347
Modified age          0
Checkpoint age        0④
0 pending log writes, 0 pending chkp writes
9 log i/o's done, 0.00 log i/o's/second
Log tracking enabled
Log tracked up to   10145937666
Max tracked LSN age 80826164
```

- Innodb_lsn_current ①
- Innodb_lsn_flushed ②
- Innodb_lsn_last_checkpoint ③
- Innodb_checkpoint_age ④
- Innodb_checkpoint_max_age ⑤

5. 与缓冲池相关的状态变量

XtraDB 将 SHOW ENGINE INNODB STATUS 命令的执行结果中有关 BUFFER POOL AND MEMORY 部分的状态值注册为 9 个新的状态变量。

```
BUFFER POOL AND MEMORY
----------------------
Total memory allocated 137363456 ③; in additional pool allocated 0
Total memory allocated by read views 88 ⑦
Internal hash tables (constant factor + variable factor)
    Adaptive hash index 2266736          (2213368 + 53368) ①
    Page hash           139112 (buffer pool 0 only)
    Dictionary cache    729463  (554768 + 174695)
    File system         824800  (812272 + 12528)
    Lock system         333248  (332872 + 376)
    Recovery system     0       (0 + 0)
Dictionary memory allocated 174695 ②
Buffer pool size        8191
Buffer pool size, bytes 134201344
Free buffers            7481
Database pages          707
Old database pages      280 ⑥
Modified db pages       0
Pending reads 0
Pending writes: LRU 0, flush list 0 single page 0
Pages made young 0 ⑤, not young 0 ④
0.00 youngs/s, 0.00 non-youngs/s
Pages read 707, created 0, written 1
0.00 reads/s, 0.00 creates/s, 0.00 writes/s
No buffer pool page gets since the last printout
Pages read ahead 0.00/s, evicted without access 0.00/s, Random read ahead 0.00/s
LRU len: 707, unzip_LRU len: 0
```

- Innodb_mem_adaptive_hash ①
- Innodb_mem_dictionary ②
- Innodb_mem_total ③
- Innodb_buffer_pool_pages_LRU_flushed

- Innodb_buffer_pool_pages_made_not_young ④（为了监测而新添加的状态值）
- Innodb_buffer_pool_pages_made_young ⑤
- Innodb_buffer_pool_pages_old ⑥
- Innodb_descriptors_memory
- Innodb_read_views_memory ⑦（为了监测而新添加的状态值）

6. 与事务相关的状态变量

XtraDB 将 SHOW ENGINE INNODB STATUS 命令的执行结果中有关 TRANSACTIONS 部分的状态值注册为 6 个新的状态变量。

```
TRANSACTIONS
------------
Trx id counter F561FD
Purge done for trx's n:o < F561EB ③ undo n:o < 0 ④
History list length 19 ①
LIST OF TRANSACTIONS FOR EACH SESSION:
---TRANSACTION 0, not started, process no 993, OS thread id 140213152634640
mysql thread id 15933, query id 32109 localhost root
show innodb status
---TRANSACTION F561FC ②, ACTIVE 29 sec, process no 993, OS thread id 140213152769808
updating or deleting
mysql tables in use 1, locked 1 ⑤
```

- Innodb_history_list_length ①
- Innodb_max_trx_id ②
- Innodb_oldest_view_low_limit_trx_id ②
- Innodb_purge_trx_id ③
- Innodb_purge_undo_no ④
- Innodb_current_row_locks ⑤

6.3.6 XtraDB 重做日志存档

XtraDB 与 InnoDB 会循环使用重做日志，并覆盖已使用的区域以便再次使用。因此，旧的重做日志会被新的重做日志覆盖。XtraDB 添加了新功能，使旧的重做日志被新的重做日志覆盖之前，可以复制旧日志。使用该功能可以收集 XtraDB 数据表的所有更改作业日志。

启用 XtraDB 的日志存档功能后，针对原来的重做日志，XtraDB 会另外复制一套重做日志。需要时，还可以创建新文件。存档的日志文件名为 ib_log_archive_<起始 _LSN>，起始 LSN 指出当前存档文件从哪个日志序列号（Log Sequence Number）开始。如下创建存档的日志文件名。

```
ib_log_archive_00000000010145937920
```

```
ib_log_archive_00000000010196267520
```

为了强制删除存档的重做日志，添加了另外的管理员命令。下列两条命令都需要使用SUPER 权限。

PURGE ARCHIVED LOGS BEFORE

删除指定日期以前创建的重做日志的存档文件，不删除当前正在复制的重做日志存档。

PURGE ARCHIVED LOGS TO

将参数指定的 log_filename 文件及其之前创建的所有重做日志的存档文件全部删除，不删除当前正在复制的重做日志存档。

注意 对重做日志进行存档时，都需要针对原来的重做日志文件以及存档的重做日志文件进行写操作，所以需要更多磁盘 I/O。另外，重做日志的空闲空间不足时，可能引发更严重的磁盘刷新行为。

XtraDB 添加了如下几个系统变量，用于控制是否启用重做日志存档，以及何时删除存档的日志。

innodb_log_archive

通过设置该变量为 ON 或 OFF，可以打开或关闭重做日志的存档功能。

innodb_log_arch_dir

设置存档的重做日志文件存储的目录。若不设置任何值，则默认存档至 MySQL 数据目录。

innodb_log_arch_expire_sec

若存档的重做日志文件一直累积，可能造成磁盘空间不足。要为此设置时间，超过该时间时，就自动删除存档的重做日志文件。该变量的默认值为 0，表示不使用自动删除功能。

重做日志的存档功能是为 XtraBackup 的增量备份（Incremental Backup）而引入的。引入重做日志的存档功能前，XtraBackup 进行增量备份时必须搜索所有数据文件以查找修改的页面，而且最后增量备份之后的数据也不容易恢复，现在，利用存档的重做日志可以将数据恢复到服务器崩溃之前的状态。

6.3.7 跟踪修改页面 MariaDB

XtraDB 添加了基于重做日志条目的修改页面跟踪功能。页面修改历史记录在特殊的位图文件中，借助这些页面的修改历史，XtraBackup 增量备份不再需要对数据文件的所有页面进行扫描以查找修改的页面，所以它对于提升增量备份的性能具有划时代的意义。通过 innodb_track_changed_pages 系统变量可以打开或关闭修改页面的历史管理功能。

所跟踪页面的修改信息记录在位图文件中，位图文件的命名遵从 ib_modified_log_<seq>_<startlsn>.xdb 的格式。<seq> 是位图日志文件的序列号，<startlsn> 是当前文件开始跟踪的日志序列号。位图文件命名采用如下格式。

```
ib_modified_log_1_0.xdb
ib_modified_log_2_1603391.xdb
```

位图文件名称中的序列号用于简单检查所需位图文件是否正常存在。查询 XtraBackup 与 INFORMATION_SCHEMA 的数据表时，日志序列号用于选择所需日志序列号范围的文件。每当服务器重启或超过所定义的最大文件大小时，位图文件就要轮换，位图文件的最大大小由 innodb_max_bitmap_file_size 系统变量设置。

如上所述，位图文件用于存储修改页面的跟踪信息。为了管理这些位图文件，添加了如下几种管理命令。

FLUSH CHANGED_PAGE_BITMAPS

该命令用于将当前重做日志检查点的修改页面同步到磁盘的位图文件。XtraBackup 需要引用位图文件时，执行该命令。

RESET CHANGED_PAGE_BITMAPS

删除所有位图文件，初始化位图文件序列号（<seq>）。

PURGE CHANGED_PAGE_BITMAPS BEFORE<lsn>

删除指定日志序列号之前的所有修改页面历史。

此外，SHOW ENGINE INNODB STATUS 命令的结果中也显示了修改页面的跟踪状态。SHOW ENGINE INNODB STATUS 的 LOG 部分中，“Log tracked up to:”项目显示依据日志跟踪线程（Log tracking thread），哪个日志序列号之前的页面修改历史会保存到磁盘。LOG 部分的“Max tracked LSN age:”项目显示页面修改历史的跟踪与当前日志序列号间所允许的最大延迟（Delay）。

通过 INFORMATION_SCHEMA 数据库的 INNODB_CHANGED_PAGES 内存表也可以访问修改页面的跟踪信息。借助 innodb_max_changed_pages 系统变量可以设置加载多少 INNODB_CHANGED_PAGES 内存表的内容。该系统变量的默认值为 1000 000，设置过大将会增加内存需求。如果无特别修改需求，建议采用默认值。

6.4 全文检索引擎 MariaDB MySQL

MySQL 5.5 中，只有使用 MyISAM 存储引擎的数据表才能使用全文检索索引。但 MyISAM 自身架构的问题导致其不如 InnoDB 存储引擎稳定。与 InnoDB 存储引擎相比，MyISAM 存储引擎不支持事务或记录行级锁等优化功能，所以实用价值不太大。因此，MySQL 5.6 的 InnoDB 存储引擎添加了可以使用全文检索引擎的功能。

6.4.1 添加全文检索索引

下面介绍 MySQL 5.6 InnoDB 添加的全文检索引擎的使用方法与特性。首先创建用于全文

检索的数据表 employee_name_innodb，它包含 emp_no、first_name、last_name 三个数据列，数据表的数据来自 employees 数据表。

```
CREATE TABLE employee_name_innodb (
  emp_no int(11) NOT NULL,
  first_name varchar(14) NOT NULL,
  last_name varchar(16) NOT NULL,
  PRIMARY KEY (emp_no)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

mysql> INSERT INTO employee_name_innodb
       SELECT emp_no, first_name, last_name FROM employees;
Query OK, 300024 rows affected (2.74 sec)
Records: 300024  Duplicates: 0  Warnings: 0
```

上面创建的数据表使用了 InnoDB 存储引擎，为了进行测试，有意不创建全文检索引擎。从 employees 数据表选出（SELECT）emp_no、first_name、last_name 三个数据列，插入（INSERT）employee_name_innodb 数据表。下面为该数据表创建全文检索引擎。

```
mysql> ALTER TABLE employee_name_innodb
            ADD FULLTEXT INDEX fx_firstname_lastname (first_name, last_name);
Query OK, 0 rows affected, 1 warning (3.70 sec)
Records: 0  Duplicates: 0  Warnings: 1

mysql> SHOW WARNINGS;
+---------+------+-----------------------------------------------------+
| Level   | Code | Message                                             |
+---------+------+-----------------------------------------------------+
| Warning |  124 | InnoDB rebuilding table to add column FTS_DOC_ID |
+---------+------+-----------------------------------------------------+
1 row in set (0.00 sec)
```

使用 ALTER TABLE 命令生成全文检索引擎索引时，出现 1 条警告信息。查看警告信息可知，由于数据表中无 FTS_DOS_ID 数据列，所以对数据表进行了重建以添加。其实，FTS_DOS_ID 数据列对用户是隐藏的，在内部只用于连接全文检索索引和数据表的记录。若原数据表中已经存在 AUTO_INCREMENT 创建的主键，仍要新添加 FTS_DOC_ID 数据列。无任何全文检索索引的数据表中，首次创建全文检索索引时，就会如上进行数据表重建，并添加 FTS_DOC_ID 数据列。向已有全文检索索引的数据表添加新的全文检索索引时，会重新使用原有的 FTS_DOC_ID 数据列，故不需重建。

前面 MySQL 5.6 的在线 Schema 修改（Online DDL）中已经提到过，不能向数据表在线添加第一个全文检索索引就是因为添加了 FTS_DOC_ID 数据列。由于向数据表添加首个全文检索索引的代价非常高，且无法在线操作，所以若需要，请在服务开始前就进行创建。

6.4.2 全文检索索引表空间

向 InnoDB 数据表添加全文检索索引后，就会生成如下几个 *.ibd 文件。

```
employee_name_innodb.ibd
FTS_0000000000000093_0000000000000153_DOC_ID.ibd
FTS_0000000000000093_ADDED.ibd
FTS_0000000000000093_BEING_DELETED.ibd
FTS_0000000000000093_BEING_DELETED_CACHE.ibd
FTS_0000000000000093_CONFIG.ibd
FTS_0000000000000093_DELETED.ibd
FTS_0000000000000093_DELETED_CACHE.ibd
FTS_0000000000000093_STOPWORDS.ibd
```

employee_name_innodb.ibd 文件是 InnoDB 存储引擎存储表数据的默认表空间。除此之外，其他各文件作用如下。

FTS_xxx_yyy_DOC_ID.ibd

全文检索索引使用停止词（StopWord）切分列值，并分解为单个检索词，从而创建索引。FTS_xxx_yyy_DOC_ ID.ibd 表空间是由单个检索词形成的索引表空间。

FTS_xxx_CONFIG.ibd

存储全文检索索引的设置内容，比如 optimize_checkpoint_limit、synced_doc_id、deleted_doc_count、stopword_table_name、use_stopword 等，它们大部分用于优化 InnoDB 全文检索索引的性能或进行调试。

FTS_xxx_STOPWORDS.ibd

MySQL 5.6 InnoDB 全文检索以指定的停止词单元切分指定文档内容，并创建索引。此时，停止词可以是已经内置于 MySQL 服务器的停止词集，也可以由用户直接指定。FTS_xxx_STOPWORDS.ibd 表空间保存相关全文检索索引使用的停止词。

FTS_xxx_ADDED.ibd 与 FTS_xxx_DELETED.ibd

MySQL 5.6 的 InnoDB 全文检索索引中，用户增删记录时，相应内容不会立即反映到全文检索索引。不同特性的数据会有所不同，但若存储在全文检索索引目标列中的数据很大，则一个数据列就有可能产生数百个乃至数千个检索字。将这些内容实时反映到全文检索索引将会是个相当大的负担。因此，InnoDB 的全文检索引擎中，添加或删除记录时，只将修改的记录内容保存到 FTS_xxx_ADDED.ibd 与 FTS_xxx_DELETED.ibd 文件，以后再对实际全文检索索引进行修改调整。使用全文检索索引进行检索时，首先从全文检索索引搜索检索字，并以 FT_DOC_ID 为基准进行合并，在得到的结果中浏览 FTS_xxx_ DELETED.ibd 文件，去除删除的记录，最后返回给用户。

带有全文检索索引的数据表经常变动时，需注意：若数据表的记录一直更改，数据会一直在 FTS_xxx_ADDED.ibd 与 FTS_xxx_DELETED.ibd 文件中累积。若想将这些内容全部反映到全文检索索引，就要使用 ALTER TABLE employee_name_innodb ENGINE=InnoDB 或 OPTIMIZE TABLE employee_name_innoDB 等命令重建数据表。

FTS_xxx_BEING_ADDED.ibd 与 FTS_xxx_BEING_DELETED.ibd

使用 OPTIMIZE TABLE employee_name_innoDB 命令重建带有全文索引的数据表，FTS_xxx_ADDED.ibd 与 FTS_xxx_DELETED.ibd 文件内容就会应用到主全文索引。此时，InnoDB 存储引擎在应用到主全文索引过程中，会将正在处理记录的 FTS_DOC_ID 值临时存储到 FTS_xxx_BEING_ADDED.ibd 与 FTS_xxx_BEING_DELETED.ibd 文件。若瞬间完成向主全文检索索引的应用处理，则两个文件通常为空，主要用于调试。

FTS_xxx_ADDED_CACHE.ibd 与 FTS_xxx_DELETED_CACHE.ibd

对带有全文检索索引的数据表进行 INSERT 或 DELETE 操作时，InnoDB 存储引擎不会直接将新列值应用到全文索引，而是将记录保存到与主全文检索索引分隔的临时全文索引，此时使用 FTS_xxx_ADDED_CACHE.ibd 与 FTS_xxx_DELETED_CACHE.ibd 文件。

如果每次执行 INSERT 或 DELETE 操作都将相应内容应用到全文检索索引，就会消耗大量系统资源。因此，InnoDB 全文检索引擎中，全文检索索引只创建 1 次，之后一直保持不变。对于后面插入或删除的记录，则另外创建临时全文索引进行管理。图 6-20 表示 InnoDB 全文检索引擎的二元化全文检索索引管理方法。若使用 ALTER TABLE 或 OPTIMIZE TABLE 命令重建数据表，则创建的临时索引就会合并到主（Main）全文索引。如果插入或删除的记录一直累积，那么临时全文索引就可能变得相当大。此时，定期重建数据表会对提升全文检索索引的性能有很大帮助。

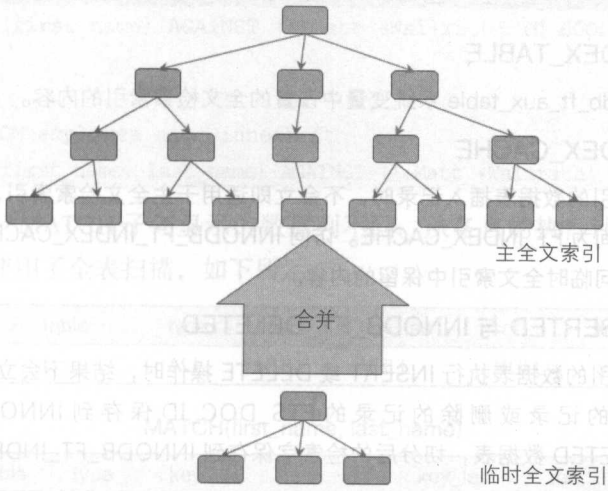


图 6-20 临时全文索引的用法

上述数据文件中，FTS_xxx_yyy_DOC.ID.ibd 数据文件必须根据全文检索索引的类别逐一存在。除此之外，其他数据文件在不同时间点有可能存在，也有可能不存在。若数据表中只删除全文索引，则会删除 FTS_xxx_yyy_DOC_ID.ibd 表空间，但不会删除其他表空间。如果想删除其余所有表空间文件，只需重建数据表。

6.4.3 与全文检索索引相关的 INFORMATION_SCHEMA 信息

如前所述，创建全文检索索引时要单独准备多个表空间。但除了用于存储 employeename innodb 数据表记录的 employee_name_innodb.ibd 表空间之外，其他表空间都无法显式访问。因此，InnoDB 在 INFORMATION_SCHEMA 数据表中提供了与上面各个表空间对应的数据表。

1. 适用于 InnoDB 的所有全文检索索引的内容

INNODB_FT_CONFIG

保存全文检索索引的基本设置。

INNODB_FT_DEFAULT_STOPWORD

全文检索索引以检索字为单位建立索引，检索字由停止词分割文档内容而成。此时，可以使用 MySQL 5.6 InnoDB 中提供的默认停止词，也可以使用用户单独创建的停止词表。在 INNODB_FT_DEFAULT_STOPWORD 表中可以查看 InnoDB 的默认停止词列表。令人遗憾的是，该停止词表的默认值只适用于英文。

2. 适用为带有全文检索索引的数据表单位的内容

INFORMATION_SCHEMA 以表为单位进行适用，设置 innodb_ft_aux_table 系统变量时，目标数据表的数据库名及表名都要明确指出，数据库名与表名以“/”分隔。

```
mysql> SET GLOBAL innodb_ft_aux_table='employees/employee_name_innodb';
```

INNODB_FT_INDEX_TABLE

可以直接访问 innodb_ft_aux_table 系统变量中设置的全文检索索引的内容。

INNODB_FT_INDEX_CACHE

向带有全文检索索引的数据表插入记录时，不会立即适用于主全文检索索引，而是保存到临时空间，此时使用的临时空间为 FT_INDEX_CACHE。访问 INNODB_FT_INDEX_CACHE 不会适用于主全文检索索引，并可以访问临时全文索引中保留的内容。

INNODB_FT_INSERTED 与 INNODB_FT_DELETED

对带有全文检索索引的数据表执行 INSERT 或 DELETE 操作时，结果不会立即适用于主全文检索索引。此时新添加的记录或删除的记录的 FTS_DOC_ID 保存到 INNODB_FT_INSERTED 或 INNODB_FT_DELETED 数据表，切分后的检索字保存到 INNODB_FT_INDEX_CACHE 数据表。

INNODB_FT_BEING_INSERTED 与 INNODB_FT_BEING_DELETED

通过 OPTIMIZE TABLE 或 ALTER TABLE 命令重建数据表时，INNODBFT_INSERTED 或 INNODB_FT DELETED 数据表的内容会适用于主全文检索索引。此时，INNODB_FT_BEING_INSERTED 与 INNODB_FT_BEING_DELETED 两个数据表用于保存正在处理的记录信息。

6.4.4 使用全文检索索引

与 MyISAM 存储引擎一样，使用 InnoDB 存储引擎的全文检索索引也要使用全文检索专用查询语法。

```
mysql> SELECT * FROM employee_name_innodb
       WHERE MATCH(first_name, last_name)
             AGAINST ('+Matt +Wallrath' IN BOOLEAN MODE);

+--------+------------+-----------+
| emp_no | first_name | last_name |
+--------+------------+-----------+
|  13507 | Matt       | Wallrath  |
+--------+------------+-----------+
```

全文检索查询中有一点需要注意，MATCH 子句必须明确罗列组成全文检索索引的全部数据列。由于 employee_name_innodb 数据表的 fx_firstname_lastname 全文索引由 first_name、last_name 两个数据列组成，所以 MATCH 子句一定要明确显示 (first_name, last_name)（与顺序无关）。

```
mysql> EXPLAIN
       SELECT * FROM employee_name_innodb ft
       WHERE MATCH(first_name) AGAINST ('+Matt +Wallrath' IN BOOLEAN MODE);

mysql> EXPLAIN
       SELECT * FROM employee_name_innodb ft
       WHERE MATCH(first_name, last_name) AGAINST ('+Matt +Wallrath' IN BOOLEAN MODE);
```

上述两个查询中，MATCH 子句显示的数据列不同。从各自的执行计划看，未列出全文索引所有数据列的查询使用了全表扫描，如下所示。

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	ft	ALL	NULL	NULL	NULL	300024	Using where

MATCH(first_name, last_name)

id	select_type	table	type	key	key_len	ref	rows	Extra
1	SIMPLE	ft	fulltext	fx_firstname_lastname	0		1	Using where

6.4.5 注意事项

MySQL 5.5 中，将带有全文索引的 MyISAM 数据表转换为 InnoDB 时，有几点需要注意。首先，MyISAM 数据表的全文检索索引与 MySQL 5.6 的 InnoDB 全文检索索引彼此使用的停止词列表不同。InnoDB 的停止词列表个数比 MyISAM 少，由于其在技术文档中通常表示短单词，

故此更改乃有意为之。因此，在MyISAM全文索引中检索的内容将无法在InnoDB全文检索索引中检索，或MyISAM结果中可能出现不曾出现的内容。为了解决该问题，InnoDB全文检索引擎的停止词列表必须设置一致。

访问INFORMATION_SCHEMA.innodb_ft_default_stopword数据表可以查看InnoDB的基本停止词列表。若各位想直接设置适合各自服务的停止词，先要创建与innodb_ft_default_stopword数据表具有相同结构的数据表，再插入停止词，然后用“db_name/table_ name”形式设置innodb_ft_server_stopword_table系统变量，这样即可设置好数据表。当然，这样注册的新停止词列表仅适用于注册以后创建的数据表。

```
mysql> CREATE TABLE stopword_for_employees(
         value varchar(18) NOT NULL DEFAULT ''
       ) ENGINE=InnoDB;

mysql> INSERT INTO stopword_for_employees
         SELECT * FROM INFORMATION_SCHEMA.INNODB_FT_DEFAULT_STOPWORD;

mysql> SELECT * FROM stopword_for_employees limit 10;
+-------+
| value |
+-------+
| a     |
| about |
| an    |
| are   |
| as    |
| at    |
| be    |
| by    |
| com   |
| de    |
+-------+
```

stopword_for_employees数据表是employee_name_innodb数据表中要使用的停止词列表，先创建，然后复制InnoDB的基本停止词列表并插入其中。下面将stopword_for_employees数据表设置为默认停止词。

```
mysql> SET GLOBAL innodb_ft_server_stopword_table='employees/stopword_for_employees';

ERROR 1231 (42000): Variable 'innodb_ft_server_stopword_table' can't be
set to the value of 'employees/stopword_for_employees'
```

将stopword_for_employees数据表注册为InnoDB的默认停止词时，发生了错误。查看

MySQL 服务器的错误日志文件，可以看到如下错误信息："停止词列表数据表只能拥有 1 个 VARCHAR 类型的数据列。"

```
InnoDB: invalid column type for stopword table employees/stopword_for_
employees. Its first column must be of varchar type
```

令人遗憾的是，存储停止词列表的数据表只能使用 Latin1 字符集。也就是说，无法将中文、韩文、日文这类语言注册为停止词。下面将 stopword_for_employees 数据表变换为 Latin1 字符集，并注册为停止词列表数据表。

```
mysql> ALTER TABLE stopword_for_employees CONVERT TO CHARACTER SET Latin1;

mysql> SET GLOBAL innodb_ft_server_stopword_table='employees/stopword_for_employees';
Query OK, 0 rows affected (0.00 sec)
```

从执行结果可以看到，stopword_for_employees 数据表正常注册为 MySQL 服务器的默认停止符列表。InnoDB 全文检索索引中，除了 innodb_ft_server_stopword_table 系统变量外，也可以使用 innodb_ft_user_stopword_table 系统变量将各用户的停止词列表注册为数据表。若设置了 innodb_ft_user_stopword_table 系统变量，则由停止词列表数据表使用，否则使用 innodb_ft_server_stopword_table 中设置的数据表；若两个系统变量都未设置，则使用内建的默认停止词列表。

从 MyISAM 迁移到 InnoDB 全文检索时，还有一个问题需要注意，那就是单个检索字的最小长度。MyISAM 全文检索中，默认检索字的最小长度为 4，最大长度不限。ft_min_word_len 与 ft_max_word_len 两个系统变量分别用于为单个检索字指定最小及最大长度。但 InnoDB 的全文检索则使用 innodb_ft_min_token_size 与 innodb_ft_max_token_len 两个系统变量。innodb_ft_min_token_size 默认值为 3，innodb_ft_max_token_len 为 64。若原来使用 MyISAM 全文检索索引，那么在 InnoDB 全文检索中也进行相同设置后，能够保证良好的兼容性。

除此之外，InnoDB 全文检索中还新增了 innodb_ft_result_cache_limit 与 innodb_ft_total_cache _size 两个系统变量。

innodb_ft_result_cache_limit

全文检索查询的中间及最终结果首先会临时存储到 MySQL 服务器的内存，innodb_ft_result_cache_limit 系统变量用于限制该临时内存空间的大小。该系统变量的默认值为 2 GB，执行全文检索查询时，若所需的内存空间超过了 innodb_ft_result_cache_limit 变量设置的值，就会发生错误并终止查询。

innodb_ft_total_cache_size

前面已经学过，插入全文检索数据表的新记录所拥有的检索字不会立即适用于主全文索引，而是保存到临时全文索引。InnoDB 的全文检索将该临时空间称为 "Cache"，若当前 MySQL 服务器中所有全文检索索引的临时全文检索索引使用的空间超过 innodb_ft_total_cache_size 系统变量的设置值，则会强制调用同步逻辑。

注意 全文检索有多种检索算法，最具代表性的有MySQL中使用的停止词方式以及n-Gram方式。MySQL 5.6 InnoDB的全文检索引擎与原来MySQL 5.5的MyISAM数据表全文检索引擎一样，只能使用停止词方式。停止词方式是指，使用指定的停止词切分文档内容，创建检索关键字，再生成关键字B-Tree索引。这种形式的全文检索仅用于关键字一致的检索，不适用于检索与空格或句子符号无关并含有特定单词的文档。若要使用MySQL服务器的全文检索功能，一定要进行详细测试，检查是否可以满足所需条件。

6.5 Memcached插件 MySQL

MySQL服务器通常用于处理包含连接或子查询的复杂形态业务条件的查询，但最近常用于基于网页的服务，也用于频繁处理简单形态的查询，比如借助主键检索1条记录。执行复杂形式的查询时，查询分析或优化的负担相对不重，但每秒执行几万次通过主键的检索查询时，查询分析与优化的负荷就变得相对更重。因此，MariaDB 5.5提供了类似HandlerSocket（https://mariadb.com/kb/en/handlersocket）的NoSQL接口。MySQL 5.6 InnoDB添加了相关功能，借助Memcached API可以读写使用InnoDB存储引擎的数据表。

注意 MySQL 5.6的Memcached插件当前只能用于Unix或Linux系列的操作系统，Windows操作系统无法使用Memcached插件。要想在MySQL 5.6中使用Memcached插件，需要用到libmemcached源代码，而libmemcached代码仍然不支持Windows操作系统。

6.5.1 架构

MySQL服务器对所有数据表的读写操作都通过Handler API层进行处理。但由于Handler API是所有存储引擎共同的API，所以在InnoDB中无法以最佳状态工作。实现Memcached插件时就决定使其不经由Handler API而直接调用InnoDB存储引擎。Memcached插件并非以独立于MySQL服务器的进程形式启动，它在MySQL服务器进程内以独立线程的形式提供服务。因此，不能用独立于MySQL服务器的硬件或进程启动Memcached插件。图6-21表示MySQL 5.6的Memcached插件的模块栈。

MySQL 5.6的Memcached插件拥有如下功能与特点。

- 以Memcached MySQL服务器插件形式运行，且运行于与MySQL服务器相同的进程中，所以对InnoDB存储引擎的访问速度相当快。
- 借助InnoDB API，对查询分析、优化、Handler API进行快速迂回（bypass）处理。
- 支持基于Memcached文本及二进制的协议，Memcapable的55个兼容测试通话。
- 可将InnoDB数据表的多列用作停止词并映射为Memcached的Value。
- 可将Memcached的缓存内存与InnoDB缓冲池内存组合使用。
- 可使用daemon_memcached_option系统变量并设置Memcached服务器选项。
- 可使用innodb_api_trx_level系统变量并设置流入Memcached插件的请求隔离级别。

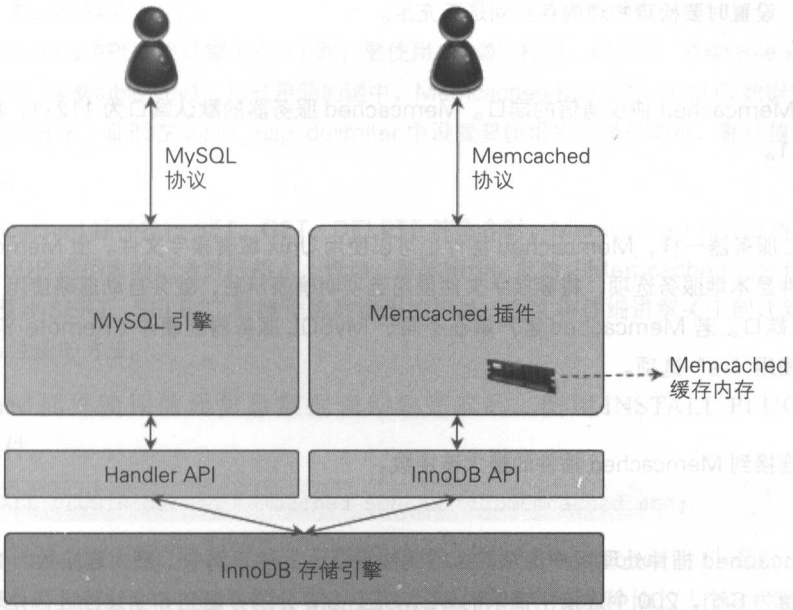


图 6-21 Memcached 插件的工作方式

6.5.2 安装与测试

下面简单介绍有关 MySQL 5.6 的 Memcached 插件安装与测试的方法。Memcached 插件内部使用 libevent 库，所以首先要安装 1.4.3 以上版本的 libevent 库。安装像 RPM 一样的打包版本时，安装 libevent 库即可。但直接编译 MySQL 服务器源码时，就要安装 libevent-devel 包。直接进行编译时，只有向 cmake 选项添加“-DWITHINNODB MEMCACHED=ON”选项，才能一起生成 InnoDB API 插件（innodb_engine.so）与 Memcached 插件共享库（libmemcached.so）。

MySQL 服务器安装完毕后，要修改 my.cnf 设置文件，以使 Memcached 插件随 MySQL 服务器一起启动。与 Memcached 插件有关的系统变量有很多，首先介绍用于启动 Memcached 的基本选项。

```
## 使用 TCP/IP 端口
daemon_memcached_option = '-m 64 -p 11211 -c 200 -t 6'

## 使用 Unix/Linux 套接字文件
daemon_memcached_option = '-m 64 -c 200 -t 6 -s /tmp/memcached.sock'
```

下面看看 Memcached 插件选项中的几个重要选项。

-m

指定 Memcached 插件本身使用的本地缓存内存的大小，单位为 MB。上述示例中，缓存内存设置为 64 MB。此处设置的本地缓存内存在分配时与 MySQL 服务器的查询缓存、主缓冲、InnoDB 缓冲池

完全无关，设置时要检查物理内存空间是否充足。

-p

指定用于 Memcached 协议通信的端口。Memcached 服务器的默认端口为 11211，所以上述示例设置为 11211。

-s

与 MySQL 服务器一样，Memcached 插件也可以使用 Unix 域套接字文件。但 Memcached 插件中，套接字文件是本地服务选项，将套接字文件使用选项明确表示后，就会自动忽略使用“-p”选项指定的 TCP/IP 端口。若 Memcached 客户端在不同于 MySQL 服务器的硬件（Remote Server）上运行，则不可以使用“-s”选项。

-c

指定可以连接到 Memcached 插件的最大连接数。

-t

指定 Memcached 插件处理用户请求时运行的线程数。上述示例中，最大连接数为 200 个，但服务器线程设置为 6 个，200 个连接中请求的 SET/GET 命令分别分配到 6 个线程进行处理。这与为每个客户端连接创建 1 个单独线程进行处理的 MySQL 服务器具有不同的操作结构。

-v、-vv、-vvv

对于 Memcached 插件运行时发生的事件或错误日志，可以使用 -v、-vv、-vvv 选项决定是否将其保存到文件。-v 选项只记录管理员必须知道的错误日志，-vv 选项将错误与警告信息等一起输出，-vvv 选项将详细诊断内容一起输出到日志文件。Memcached 插件无专用的日志文件，Memcached 插件日志会记录到 MySQL 服务器的错误日志文件。

Memcached 插件安装完毕，下面学习如何在 MySQL 5.6 中激活。启动 MySQL 服务器，输入如下命令，创建 Memcached 插件运行所需的元数据表。

```
mysql> SOURCE ${MYSQL_HOME}/share/innodb_memcached_config.sql
```

上面生成 Memcached 元数据表的脚本总共创建了 innodb_memcache 数据库以及 3 个元数据表，各数据表的用途如下。

cache_policies

该数据表用于设置有关 Memcached 的 SET 与 GET、DELETE 及 FLUSH 的缓存策略。上述初始化脚本默认创建了名为 cache_policy 的缓存策略，它不使用 Memcached 的缓存内存，直接对 InnoDB 数据表进行 INSERT 与 SELECT 操作。若想更改缓存策略，只要直接更新（UPDATE）该数据表列即可。后文将进一步讲解关于缓存策略的详细内容。

config_options

该数据表用于保存变换等设置，使 InnoDB 数据表的记录可以使用 Memcached 协议进行传送。MySQL 5.6.15 支持 separator 与 table_map_delimiter 两个设置。

- 将 InnoDB 数据表的多个列映射为 Memcached 的值（Value）时，separator 选项设置各数据列的

分隔符。默认分隔符为“|”。

- 在 Memcached API 中声明键（Key）时，要使用“@@”标记，即采用“@@table.key1”形式指定键值对。以“@@table.key1”形式声明的键中，Memcached 插件要将 InnoDB 数据表名与实际键值对的键分离开来，此时在 table_map_delimiter 中设置要使用的分隔符即可，默认值分隔符为“.”。

containers

通过 Memcached 插件执行 SET、GET、DELETE 等命令时，Memcached 插件会再次通过 InnoDB API 对 InnoDB 数据表的记录进行插入、查询、删除操作。此时，Memcached 插件使用 containers 数据表管理 INSERT、SELECT 要操作的数据表的信息。后文将详细讲解关于创建新数据表以及向 containers 注册的方法。

Memcached 插件使用的元信息数据表创建完成后，使用 INSTALL PLUGIN 命令激活 Memcached 插件。

```
mysql> INSTALL PLUGIN daemon_memcached SONAME "libmemcached.so";
```

使用 Memcached 插件的所有准备工作到此结束。测试 Memcached 插件能否正常工作之前，先要知道如何连接 Memcached 服务器。Memcached 服务器使用两种协议与客户端通信。

文本协议

Memcached 一直使用类似于 SMTP、HTTP 的基于文本的协议。顾名思义，文本协议就是将用户输入的内容原原本本地传送给 Memcached 服务器，服务器可以理解文本命令。基于文本的协议中，由于不需要考虑字节存储顺序（Endian），所以与其他系统进行数据交换时，字节传送顺序处理起来非常简单。

二进制协议

文本协议易于人眼识别，又有良好的系统兼容性，但必须分析输入的文本命令，基于文本的内容比二进制数据需要更多数据空间。也就是说，基于文本的协议要占用大量 CPU 及网络带宽，为解决该问题而引入二进制协议。

MySQL 5.6 的 Memcached 插件同时支持文本协议与二进制协议，下面使用基于文本的协议进行测试。使用基于文本的协议连接 Memcached 服务器时，可以使用 Telnet 程序。

```
shell> telnet localhost 11211
Trying 127.0.0.1...
Connected to localhost (127.0.0.1).
Escape character is '^]'.
```

在 Linux 或 Unix shell 中使用 telnet 命令时，最后输入 Memcached 插件的端口号 1121 后，即显示上述示例中的连接信息。Memcached 服务器处于连接状态下，输入 SET 或 GET 命令进行执行。下列示例带下划线的部分是要直接输入的命令，先输入如下命令进行测试。与 INSERT 命令一样，SET 命令将值保存到 Memcached 插件与 InnoDB 数据表。输入“set matt 0 3600 12”后，一定要输入“回车”（Enter），然后再输入“Real MariaDB”值。保存完毕后，显示

“STORED”响应信息。

```
set matt 0 3600 12
Real MariaDB
STORED
```

下面使用GET命令获取保存的数据。执行如下命令，完整显示前面使用SET命令保存的值。

```
get matt
VALUE matt 0 12
Real MariaDB
END
```

下面删除使用SET命令保存的数据。从Memcached服务器删除键值对时，只要使用DELETE命令即可。

```
delete matt
DELETED
```

向Memcached插件输入SET或GET命令时，一定要使用小写字母。如下所示，输入大写字母就会发生ERROR，并中止处理。

```
SET matt 0 3600 12
ERROR
```

测试结束后，输入“quit”命令并按Enter键，断开与Memcached服务器的连接。安装Memcached插件以使用Unix域套接字文件时，不能如上使用telnet程序进行测试，而要使用NetCat程序，如下所示。与使用telnet程序不同，使用NetCat程序时，连接之初不会输出任何信息。输入SET/GET命令可以检查是否正常连接。

```
shell> nc -U /tmp/memcached.sock
```

使用GET与DELETE命令时，只要在指令之后输入要检索或删除的键值对的键，然后按Enter键即可。但使用SET命令时，SET指令之后需要几个参数，如下所示。

<key>

要存储在Memcached服务器中的键值对的键。MySQL服务器的Memcached插件可以保存的最大键为250字节。

<flag>

flag不是Memcached服务器使用的值，应用程序有更多需要额外管理的值时，可以使用该字段。flag是4字节整数类型。

<expire_time>

Memcached中使用SET命令保存的键值对并非一直保留于内存。保存在Memcached中的键值对

遇到 Memcached 缓存内存不足或超过预约的时间时，就会自动从 Memcached 缓存内存中删除。

expire_time 设置保存的键值对的有效期，超过有效期时，就会自动从内存缓存中删除。

expire_time 可以以秒为单位保存绝对时间与相对时间。向 expire_time 输入小于 2592000（30 天）的值时，识别为相对时间，从现在开始 30 天后自动从内存缓存中删除。若输入大于 30 天的值，Memcached 服务器将 expire_time 解析为 Unix 时间戳，并认为相关键值对到该时间前始终有效。

<byte_count>

用于指明要 SET 的键值对的值共有多少字节。Memcached 服务器中最大可以保存 1 MB 的值。

<value>

输入 byte_count 指定长度的值。

6.5.3 缓存策略

如图 6-21 所示，Memcached 插件也使用自身的内存缓存。根据该内存缓存的使用方法，可以划分为 4 种缓存策略。Memcached 插件的缓存策略可以通过直接对 cache_policies 元数据表进行插入或更新操作而进行设置，对每个数据表可以设置不同的缓存策略。

innodb_only

innodb_only 缓存策略中，忽略 Memcached 插件的全部内存缓存。也就是说，通过 Memcached 插件调用 GET/SET 时，Memcached 插件会忽视 Memcached 自身的所有内存缓存，而将请求发送给 InnoDB API，再将从 InnoDB 存储引擎获取的结果原原本本地返回给用户。与执行 INSERT 或 UPDATE 查询一样，通过 Memcached 插件添加（SET）的数据会永久保存到 InnoDB 数据表。

cache_only

与 innodb_only 相反，cache_only 缓存策略只检索 Memcached 插件的内存缓存，响应用户的 GET/SET 请求。此时，在与 InnoDB 存储引擎数据表拥有的数据完全无关的环境下工作，就像只有 Memcached 服务器启动一样。由于本身没有数据要永久写到磁盘，所以 MySQL 服务器重启时，数据会自动初始化。

caching

caching 缓存策略中，InnoDB 存储引擎与 Memcached 插件协同工作。用户执行 GET 命令时，Memcached 插件首先在 Memcached 内存缓存中检索请求的数据，若查找到请求的数据，则立即将数据返回给用户；若查找不到，则向 InnoDB 存储引擎提出检索请求。用户执行 SET 命令时，Memcached 插件先将数据保存到 Memcached 的内存缓存，再通过 InnoDB API 将数据永久记录到 InnoDB 数据表。caching 缓存策略中，MySQL 服务器与 Memcached 服务器就像在不同进程中同时工作一样。

disabled

disabled 缓存策略不使用 Memcached 插件。

6.5.4 注册用户数据表

安装 MySQL 服务器后，一同发布的“${MYSQL_HOME}/share/innodb_memcached_config.sql”初始化脚本默认在 test 数据库中创建并注册了名为 test_demo 的数据表，使 Memcached 插件可以识别，所以前文才能对 Memcached 插件进行测试。Memcached 中添加或删除（SET/DELETE）的数据先保存到 test_demo 数据表中，然后删除。通过 Memcached 插件，在 MySQL 服务器中可以使用 SQL 访问保存的数据。

接下来，不针对测试脚本中自动注册的数据表，而对当前服务中正在使用的数据表或直接创建的数据表进行注册，使 Memcached 插件可以识别。对于 Memcached 插件要用作服务表或键的数据列，有如下几项限制。

- 用作键的值中不要包含空格或换行符（New-line）。因为 Memcached 的文本协议中，空格或换行符会被用作停止词。
- 用作键的值保存到数据列，该数据列只能是整型（INTERGER）或字符串型（CHAR、VARCHAR）。
- 用作键的值保存于数据列，该数据列必须创建有主键或唯一键。
- 分区表无法在 Memcached 插件中使用。

首先创建 my_service 数据库，然后在其中创建 mc_store 数据表。请注意，此处的数据库名与数据表名并不固定，可以根据服务选用合适的名称。mc_store 数据表包含 c_key、c_value、c_flag、c_cas、c_expire 等数据列，这些数据列是必须的，它们由 Memcached 插件使用。当然，除了这 5 个数据列外，还可以根据需要向数据表额外添加其他数据列。

```
mysql> CREATE DATABASE my_service;
mysql> USE my_service;

mysql> CREATE TABLE mc_store (
  c_key VARCHAR(250) COLLATE Latin1_bin NOT NULL,
  c_value MEDIUMBLOB,
  c_flag INT NOT NULL DEFAULT '0',
  c_cas BIGINT UNSIGNED NOT NULL DEFAULT '0',
  c_expire INT NOT NULL DEFAULT '0',
  PRIMARY KEY(c_key)
) ENGINE=InnoDB STATS_PERSISTENT=0;
```

然后，进入保存 Memcached 插件元信息的 innodb_memcache 数据库，并注册上面准备的 mc_store 数据表，以使 Memcached 能够识别。注册过程将决定 mc_store 数据表的各数据列在 Memcached 插件中所起的作用。

```
mysql> USE innodb_memcache;

mysql> INSERT INTO containers (
```

```
  name, db_schema, db_table, key_columns, value_columns, flags, cas_column,
  expire_time_column, unique_idx_name_on_key)
VALUES (
  'default', 'my_service', 'mc_store', 'c_key', 'c_value', 'c_flag', 'c_cas',
  'c_expire', 'PRIMARY');
```

此处需要注意的是，containers 数据表的 name 数据列插入了“default”值。其实，Memcached 中并不存在数据表的概念，但 MySQL 5.6 Memcached 插件提供了多个可以通过 Memcached 接口进行操作的数据表。也就是说，在 Memcached 接口与 MySQL 服务器的数据表间进行映射时，需要使用相应信息。为此，Memcached API 在键域部分使用“@@”标记同时标注表名与键。请看下列两个键表示示例。

```
@@product.A01
@@order.A01
```

第一个示例中，表名为 product，指代该表中键为 A01 的键值对，第二个示例表示 order 数据表的键值 A01。此时，product 与 order 都不是 InnoDB 中实际存在的数据表名，它们是注册 containers 数据表时在 name 数据列中输入的名称。若没有“@@”标记而只记为“A01”，那所指的又是哪个数据表呢？

若 containers 数据表中注册的数据表只有 1 个，Memcached 会自动将其视为唯一数据表中键为 A01 的键值对。但 containers 数据表中注册有多个 InnoDB 数据表时，所指的是以 default 名称注册的 InnoDB 数据表。若不想在 Memcached 客户端用“@@”标记指代另外的数据表，则只要在 containers 数据表的 name 数据列中存入 default 即可。

6.5.5 相关系统变量

由于 Memcached 服务器中存储的是键值对（Key/Value），所以不像 MySQL 服务器的 InnoDB 一样支持事务或隔离级别等。实际上也不需要这些概念，但 InnoDB 存储引擎中必须定义这些部分。这样就可以通过 MySQL 服务器的系统变量指定 Memcached 插件使用 InnoDB API 时的隔离级别，还可以指定将保存到 Memcached 插件的数据如何传递给 InnoDB。下面仅讲解 Memcached 插件设置中的几个重要变量。

daemon_memcached_option

MySQL 服务器初始化后，MySQL 服务器进程会以单线程形式启动 Memcached 插件。此时，MySQL 服务器会设置以哪个选项启动 Memcached 插件。

innodb_api_trx_level

通过 Memcached 插件的连接与通过 MySQL 的连接更改或引用同一条记录时，innodbapi trx_level 设置 Memcached 插件连接使用哪种隔离级别，其值为整数型，默认值为 0，即 READ_UNCOMMITTED。

- 0 READ_UNCOMMITTED
- 1 READ_COMMITTED
- 2 REPEATABLE_READ
- 3 SERIALIZABLE

daemon_memcached_r_batch_size 与 daemon_memcached_w_batch_size

通过 Memcached 插件执行 SET/GET 等命令时，以上变量用于设置执行多少 SET/GET 后提交（Commit）。若将其值设置为 10，则 Memcached 插件在各连接中必须执行 10 次 SET 后再提交。若执行 9 次 SET 命令后，MySQL 服务器异常终止（崩溃），尚未提交的 9 项数据都会丢失。该系统变量的默认值为 1。若数据不重要，可以将批量大小设置得大一些，这样有利于提高 Memcached 插件的性能。

innodb_api_bk_commit_interval

通过 Memcached 插件执行 SET、ADD、DELETE 命令时，由于使用 Memcached 协议，故没有类似 Commit 或 Rollback 的命令。这样，Memcached 插件就无法判断何时将添加或删除的键值对应用并提交到 InnoDB。若 daemonmemcached_r_batch size 与 daemon_memcached_w_batch_size 系统变量设置为 1，则每次执行 SET 命令时就会立即提交。但这样会引起 InnoDB 提交过度频繁，降低 Memcached 的性能。

将 daemon_memcached_r_batch_size 与 daemon_memcached_w_batch_size 系统变量设置为大于 1 的值，可以提高 Memcached 插件的性能。此时，命令积累到指定数量才会进行提交。某种情形下，连接将不执行 SET 或 ADD 等命令，一直处于空闲状态。针对该问题，Memcached 插件提供了相应功能，可以将不执行任何命令的（处于 Idle 状态的）连接集中起来，然后在后台统一提交。innodb_api_bk_commit_interval 系统变量用于决定后台线程收集提交的时间间隔，以秒为单位，默认值为 5 秒。

innodb_api_enable_binlog

通过 Memcached 插件执行 SET、ADD、DELETE 等命令时，这些数据会全部传递给 InnoDB 存储引擎，最后保存到磁盘的数据表。存储在 InnoDB 中的所有数据也要通过 MySQL 的复制传递给从服务器。为了将通过 Memcached 插件存储或更改的数据全部传递给 MySQL 从服务器，进入 Memcached 插件的更改内容要全部记录到二进制文件。innodb_api_enable_binlog 系统变量决定是否将 Memcached 插件更改的内容全部记录到二进制文件。

另外，daemon_memcached_r_batch_size 与 daemon_memcached_w_batch_size、innodb_api_bk_commit_interval 系统变量值设置得越大，MySQL 主服务器中更改的内容传递给从服务器的时间点越迟。

6.6 Cassandra 存储引擎 MariaDB

最近，NoSQL 成为人们关注的热点，MariaDB 10.0 添加了几种有关 NoSQL 的功能，具有代表性的有 HandlerSocket、动态列（Dynamic Column）、Memcached 插件等，此处又添加了 Cassandra 存储引擎功能，以访问 Cassandra 的数据。Cassandra 存储引擎实际的 Cassandra

Cluster 并不是在 MariaDB 服务器内部实现的，MariaDB 连接远程服务器中的 Cassandra 服务器，然后用 SQL 语法访问所需数据。也就是说，要使用 Cassandra 存储引擎，必须有已经处于使用状态的 Cassandra Cluster。

6.6.1 Cassandra

Cassandra 是最具代表性的 NoSQL 数据库，其内部采用 P2P（Peer to peer）方式进行通信，掌握集群（Cluster）内各节点的状态，并保持服务可用状态。图 6-22 表示 Cassandra 集群的形成及响应服务请求的过程。Cassandra 集群中，各 Cassandra 实例（Cassandra 进程）称为节点。

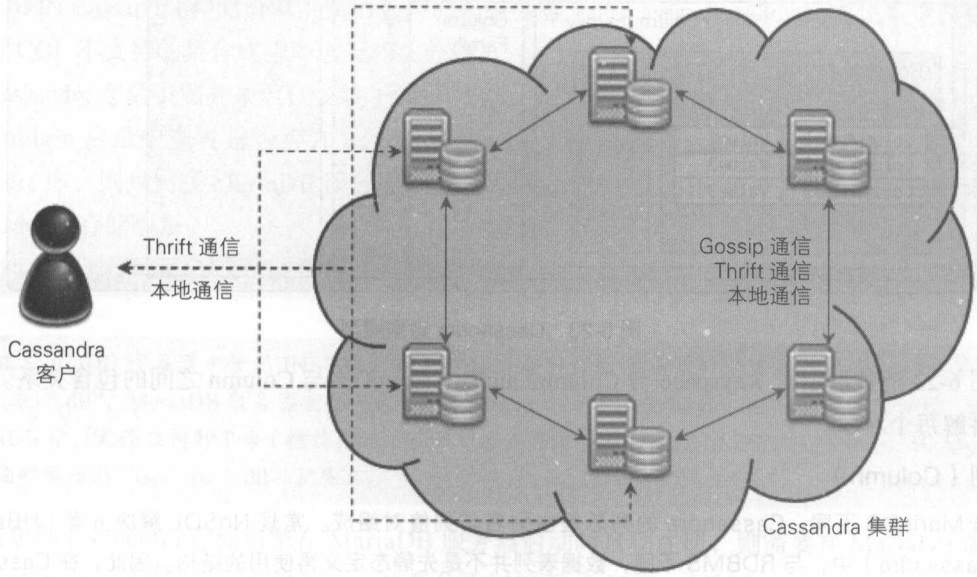


图 6-22 Cassandra 结构图

虽然使用了“P2P 方式”这种说法，但其实内部所有节点在集群中的位置是固定的，在自身位置与左右两侧的节点以及任意的几个节点保持通信，同时管理集群内所有节点的状态信息。Cassandra 节点传播其他节点状态的协议称为 Gossip 协议。各节点相互交换状态信息时，不仅传递自身状态，还将自身所知的其他节点的状态一起进行传递，就像传播谈论别人的流言蜚语一样，故使用“Gossip”一词命名。

如图 6-22 所示，Cassandra 集群内的所有节点排列成环状（Ring），每个节点都拥有全部数据的 1/N 数据，各节点只提供自身拥有的数据。若请求非自身数据，就会查找拥有请求数据的节点，然后获取所需数据并返回给客户端。也就是说，Cassandra 中的所有节点不仅提供数据，同时还在中间充当数据传递的代理。此时，各节点同其他节点或客户端进行数据交换时使用 Thrift 通信。但 Thrift 通信的缺点在于无法处理数据流，所以在最近的 Cassandra（Cassandra 1.2.x 与 2.x 版本）中编写了自身协议，该协议没有特别的称谓，通常称为“本地协议”或“二进制协议”。

与RDBMS不同，Cassandra不管理标准的数据结构，其内部以键值（Key/Value）对的形式管理数据信息。但Cassandra的Key/Value采用的不是简单的一元而是多元键值对形式进行管理，所以内部数据结构经常发生混淆。此处所说的多元Key/Value是指(Key/(Key/Value))结构，最终，用户信息不是保存在一个键中，而是以新键值对的形式保存。像这样，使用多元组成的键值对结构，Cassandra也可以组织类似于RDBMS的数据表或记录形式的数据。Cassandra的数据结构（Schema组成）如图6-23所示。

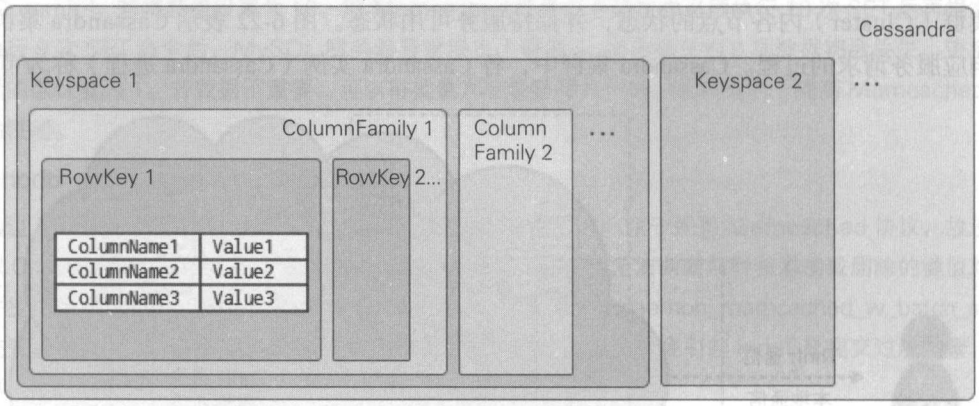


图6-23 Cassandra数据模型

图6-23清楚展示了Keyspace与ColumnFamily、RowKey与Column之间的包含关系。下面逐一讲解每个对象。

列（Column）

与MariaDB不同，Cassandra的列总是由列名与列值对组成。常规NoSQL解决方案（HBase或Cassandra）中，与RDBMS不同，数据表列并不是先静态定义再使用的结构。因此，在Cassandra中存储列时，必须将列名与列值成对保存。

RowKey

Cassandra或HBase这类NoSQL工具并不像RDBMS一样有存储标准模型的数据库，故不存在与RDBMS的Row相对应的概念，只有以“Key/Value对”进行保存的概念。HBase与Cassandra中，Cassandra与RDBMS的结构更接近一些，也提供自身SQL语法接口CQL（Cassandra Query Language）。其实，可以将Cassandra的RowKey理解为类似于RDBMS中Row的主键的对象。

ColumnFamily

ColumnFamily类似于MariaDB中的数据表，是定义数据集合的基本单位。RDBMS中，使用同一主键定义的数据（属性）全部集中于一个表并进行创建；而NoSQL解决方案中，只有经常一起使用的数据才集中起来并定义为ColumnFamily进行使用。

Keyspace

Cassandra的Keyspace类似于MariaDB的数据库，Cassandra的ColumnFamily只有在一个Keyspace内部才存在。

Cassandra 也提供了类似于 RDMBS 的 SQL 语法的 CQL 客户端。由于 CQL 客户端有如下限制，故只能用于访问非常简单的数据。

- 不支持连接和子查询。
- GROUP BY 或 ORDER BY 等集合及排序功能受限（仅限于由 RowKey+ 列名组成的列）。
- 只有 RowKey 或列名可用于 WHERE 条件。

6.6.2 Cassandra 存储引擎

之前学习了 Cassandra 的基本架构与数据模型，下面讲解 MariaDB 的 Cassandra 存储引擎。MariaDB 的 Cassandra 存储引擎主要用于为远程服务器上的 Cassandra Cluster 提供数据视图，还可以对 CQL 不支持的集合或排序进行相关处理。

Cassandra 存储引擎并未默认安装于 MariaDB，所以无法直接使用。首先在 MariaDB 主目录的 lib/plugin 目录中查看是否存在 ha_cassandra.so 文件或 ha_cassandra.dll 文件。若存在 ha_cassandra 库，则只要在 MariaDB 服务器中使用 INSTALL PLUGIN 命令进行注册，随后即可使用 Cassandra 存储引擎。

```
MariaDB> INSTALL PLUGIN cassandra SONAME 'ha_cassandra.so';
```

提示 在 MariaDB 服务器中使用 INSTALL PLUGIN 命令时，指明存储引擎库文件的部分即使不输入“.so”或“.dll”，MariaDB 服务器也会查找 ha_cassandra.so 或 ha_cassandra.dll 文件并进行加载。因此，MariaDB 中，无论在何种平台（操作系统）下都可以使用统一的 INSTALL PLUGIN 命令。而 MySQL 中必须明确指出“.so”或“.dll”扩展名。

若想使 Cassandra 存储引擎在 MariaDB 服务器启动时自动注册，则需要在 MariaDB 服务器设置文件（my.cnf 或 my.ini）的 [mysqld] 部分添加如下 plugin-load 选项。

```
plugin-load = ha_cassandra.so
```

Cassandra 存储引擎注册完毕后，下面创建 MariaDB 服务器的 Cassandra 存储引擎数据表，并访问位于远程服务器的 Cassandra 数据。首先在 Cassandra 中创建示例数据表，如下所示。

```
shell> cqlsh

cqlsh> CREATE KEYSPACE cassandra_se
   ...   WITH strategy_class = 'org.apache.cassandra.locator.SimpleStrategy'
   ...   AND strategy_options:replication_factor='1';
cqlsh> USE cassandra_se;

cqlsh:cassandra_se> CREATE COLUMNFAMILY cf_friends (
   ...                uid VARCHAR PRIMARY KEY, friend_uid VARCHAR)
   ...                WITH COMPACT STORAGE;
cqlsh:cassandra_se> INSERT INTO cf_friends (uid, friend_uid) VALUES ('matt', 'pitt');
```

```
cqlsh:cassandra_se> INSERT INTO cf_friends (uid, friend_uid) VALUES ('matt', 'pott');

cqlsh:cassandra_se> SELECT * FROM cf_friends;

 uid  | friend_uid
------+------------
 matt |       pitt
 matt |       pott
```

接下来返回 MariaDB 服务器，创建 Cassandra 存储引擎数据表。创建 Cassandra 存储引擎数据表时，Cassandra 的 ColumnFamily 与列名及数据类型必须保持一致。

```
MariaDB> CREATE TABLE cass_friends (
           uid VARCHAR(64) PRIMARY KEY,
           friend_uid VARCHAR(64)
         ) ENGINE=cassandra  THRIFT_HOST='192.168.0.1'
           KEYSPACE='cassandra_se' COLUMN_FAMILY='cf_friends';
```

若数据类型不一致或列名不对，创建数据表时就会发生错误。创建 Cassandra 存储引擎的数据表时，若使用 thrift_host 选项，则可以数据表为单位设置要连接的 Cassandra Cluster。借助 cassandra_default_thrift_host 系统变量还可以指定所有数据表共同使用的 Cassandra Cluster。cassandra_default_thrift_host 系统变量是动态变量，所以可以在不重启 MariaDB 服务器的情形下对其进行修改，使得使用 Cassandra 存储引擎的数据表可以连接到其他 Cassandra 节点。

创建 Cassandra 存储引擎数据表之后，就可以像在 MariaDB 服务器中访问普通数据表或保存数据时一样，使用 SELECT 或 INSERT 查询语句。

```
MariaDB> INSERT INTO cass_friends (uid, friend_uid) VALUES ('toto', 'pitt');
MariaDB> INSERT INTO cass_friends (uid, friend_uid) VALUES ('toto', 'pott');

MariaDB> SELECT * FROM cass_friends WHERE uid='toto';

+------+------------+
| uid  | friend_uid |
+------+------------+
| toto | pitt       |
| toto | pott       |
+------+------------+
```

Cassandra 1.1 使用 Thrift API 交换数据，但从 Cassandra 1.2 开始支持名为二进制协议的本地 API。并且从 Cassandra 1.2（含 Cassandra 2.x 版本）开始，内部数据模型逐渐发生变化，当前版本的 MariaDB 的 Cassandra 存储引擎不支持 Cassandra 1.2 版本所改善的数据模型。若想在 Cassandra 1.2 中使用与 Cassandra 1.1 数据模型兼容的结构，则必须使用 WITH COMPACT

STORAGE 选项。当前版本的 MariaDB（10.0.7 版本）的 Cassandra 存储引擎也可以激活 WITH COMPACT STORAGE 选项，仅与创建的 ColumnFamily 联动。

此外，当前版本的 MariaDB 使用 Thrift API 连接 Cassandra，其实 Thrift 带有一些限制，比如无法对数据进行流处理等。不久前，Cassandra C++ 驱动程序发布，其中好像做了一些改善，使 MariaDB 可以直接与 Cassandra 本地协议而非 Thrift 保持联动。

6.7 CONNECT 存储引擎 MariaDB

CONNECT 存储引擎提供了连接功能，使 MariaDB 可以直接访问其他——如 Oracle、MSSQL 等——DBMS 的数据表，还可以访问 XML、EXCEL 等外部文件。除此之外，CONNECT 存储引擎还帮助 MariaDB 轻松访问及引用其外部的大量数据。下面学习 MariaDB 通过 ODBC 接口访问 Oracle RDBMS 数据表及通过 CONNECT 存储引擎访问 MariaDB 设置文件（my.cnf）的方法。

请注意，在 CONNECT 存储引擎中执行 DROP TABLE 命令时，只删除了使用 CONNECT 存储引擎的数据表的定义。比如，先使用远程服务器的 Oracle RDBMS 中存在的数据表，创建为 CONNECT 存储引擎数据表，然后在 MariaDB 服务器中使用 DROP TABLE 命令删除 CONNECT 存储引擎数据表。实际远程的 Oracle 数据表未被删除，只删除了 MariaDB 服务器的数据表。但执行 UPDATE、DELETE、TRUNCATE 命令时，更改的内容会实际应用到远程服务器的数据表。

6.7.1 安装 CONNECT 存储引擎

MariaDB 不会安装某些不必要的存储引擎。MySQL 或 MariaDB 中，所有存储引擎都可以插件形式使用。若要使用尚未安装的存储引擎，只要先使用 INSTALL PLUGIN 命令注册相应插件库即可。下列命令用于在 MariaDB 中安装并使用 CONNECT 存储引擎。其实，CONNECT 存储引擎以 *.so（Linux 或 Unix 系列）文件或 *.dll（Windows）文件形式存在于 MariaDB 主目录的 lib/plugin 目录下。使用下列命令安装 CONNECT 存储引擎前，必须保证 lib/plugin 目录下存在 ha_connect.so（Unix 或 Linux）文件或 ha_connect.dll（Windows）文件。

```
MariaDB> INSTALL PLUGIN connect SONAME 'ha_connect.so';
```

CONNECT 存储引擎安装完毕后，使用 SHOW ENGINES 命令或 SHOW PLUGIN 命令可以查看加载的存储引擎列表，如下所示。

```
MariaDB> SHOW ENGINES;
+---------+---------+----------------------------------------------------------------+
| Engine  | Support | Comment                                                        |
+---------+---------+----------------------------------------------------------------+
...
```

```
| CONNECT | YES      | Management of External Data (SQL/MED), including many file formats |
...
+---------+---------+-------------------------------------------------------------------+

MariaDB> SHOW PLUGINS;
+-------------------------+----------+----------------+-----------------+---------+
| Name                    | Status   | Type           | Library         | License |
+-------------------------+----------+----------------+-----------------+---------+
...
| CONNECT                 | ACTIVE   | STORAGE ENGINE | ha_connect.dll  | GPL     |
+-------------------------+----------+----------------+-----------------+---------+
```

6.7.2 连接 Oracle RDBMS 数据表

使用 MariaDB 的 CONNECT 存储引擎可以访问 MariaDB 服务器外部的 RDBMS 服务器的数据表，大部分 RDBMS 服务器都提供通过 ODBC 进行访问的接口。在 MariaDB 中创建 ODBC 表类型的 CONNECT 存储引擎数据表后，即可访问支持 ODBC 的所有远程数据库的数据表。假设 Oracle RDBMS 中含有数据表 emp，如下所示。

```
Oracle> CREATE TABLE emp(
  empno    number(4,0),
  ename    varchar2(10),
  job      varchar2(9),
  mgr      number(4,0),
  hiredate date,
  sal      number(7,2),
  comm     number(7,2),
  deptno   number(2,0),
  CONSTRAINT pk_emp PRIMARY KEY (empno)
);
```

如下所示，在 MariaDB 服务器中创建使用 CONNECT 存储引擎的 ora_emp 数据表。所有准备工作到此结束。当然，创建 ora_emp 数据表前，先要通过 DSN 注册名为 orcl 的 ODBC 驱动。在 Windows 平台下注册并不难，若 MariaDB 服务器运行在 Linux 或 Unix 系列的操作系统中，请参考 Unix ODBC（http://www.unixodbc.org/）驱动项目的页面。

```
MariaDB> CREATE TABLE ora_emp
         ENGINE=CONNECT
         TABLE_TYPE=ODBC TABNAME='EMP' CONNECTION='DSN=orcl;UID=scott;PWD=tiger'
         SRCDEF='SELECT empno, ename, mgr, LEVEL FROM emp CONNECT BY
PRIOR empno=mgr;';
```

ora_emp 数据表准备好后，可以在 MariaDB 中使用 SELECT 查询进行访问。创建 ora_emp 数据表时，SRCDEF 选项中使用了 CONNECT BY 子句。SRCDEF 选项中的查询并不在创建 ora_emp 数据表的 MariaDB 服务器中执行，而是明确指出该查询用于在远程 Oracle RDBMS 中访问数据。因此，查询语句必须使用 Oracle RDBMS 语法。

```
MariaDB> SELECT * FROM ora_emp;
+-------+--------+------+-------+
| EMPNO | ENAME  | MGR  | LEVEL |
+-------+--------+------+-------+
|  7788 | SCOTT  | 7566 |     1 |
|  7876 | ADAMS  | 7788 |     2 |
...
+-------+--------+------+-------+
```

MariaDB 的 CONNECT 存储引擎位于远程 Oracle RDBMS 与客户端之间，只在中间提供存储空间。此时需要注意，一定要将 engine_condition_pushdown 优化器开关打开（ON）。engine_condition_pushdown 优化器开关的默认值为 OFF，处于关闭状态时，WHERE 条件句的内容不会传送到远程 RDBMS 服务器，所以不仅会降低远程 RDBMS 服务器的性能，还会导致 MariaDB 服务器性能低下。

```
MariaDB> SET optimizer_switch='engine_condition_pushdown=ON';
```

6.7.3 连接 my.cnf 设置文件

运营 MariaDB 服务器时，每次创建简单的实用程序都会感到诸多不便，其中最具代表性的是，无法轻易获取 MariaDB 服务器的设置文件。虽然使用类似 Linux shell 脚本的 GREP 命令可以轻松实现，但由于 MariaDB 服务器的设置文件采用分区形式组织（如 [mysqld] 区），且不同分区可以罗列重复选项，所以具体实现要麻烦得多。当然，MariaDB 服务器中也可以使用 SHOW GLOBAL VARIABLES 命令，但有时只使用全局变量是不够的。若想在 C/C++ 中读取 my.cnf 设置文件，只调用一两个函数绝对无法办到。为了解决该问题，可以使用 MariaDB 的 CONNECT 存储引擎，以 1 个表的形式访问 MariaDB 的设置文件。

首先，在 MariaDB 服务器中创建使用 CONNECT 存储引擎的数据表，如下所示。此处需要注意，一定要具备相应权限，使 MariaDB 服务器（启动 MariaDB 服务器进程的操作系统用户）可以访问 my.cnf 设置文件的位置。

```
MariaDB> CREATE TABLE mysql_config (
           section VARCHAR(64) flag=1,
           keyname VARCHAR(64) flag=2,
           value VARCHAR(256))
         ENGINE=CONNECT  TABLE_TYPE=INI  FILE_NAME='/tmp/my.cnf'
         OPTION_LIST='Layout=Row;seclen=90000';
```

上述示例创建了名为 mysql_config 的数据表，它有 3 个数据列，并将 TABLE_TYPE 选项设置为 INI。MariaDB 服务器的 my.cnf 文件的扩展名的确会有所不同，但内部都采用 ini Key/Value 格式。因此，在 FILE_NAME 选项中输入要连接的 my.cnf 文件的路径。TABLE_TYPE 为 INI 时，设置 OPTION_LIST 中的 Layout 选项就可以选择 Row 或 Column。Layout 为 Column 时，在 INI 文件将一个分区解析为一个数据表。以一个表记录的形式访问 Key/Value 时，通常只要将 Layout 设置为 Row 即可。srclen 选项用于设置内部缓冲大小，该缓冲存储 INI 设置文件中包含在一个分区的所有内容。

下面使用 SQL 访问 mysql_config 数据表。

```
MariaDB> SELECT * FROM mysql_config;
+-------------+----------+------------------+
| section     | keyname  | value            |
+-------------+----------+------------------+
| mysqld      | port     | 3306             |
| mysqld      | socket   | /tmp/mysql.sock  |
| mysqld      | tmpdir   | /tmp             |
| client      | port     | 3306             |
...

MariaDB> SELECT * FROM mysql_config WHERE section='mysqld' AND keyname='port';
+-------------+----------+------------------+
| section     | keyname  | value            |
+-------------+----------+------------------+
| mysqld      | port     | 3306             |
+-------------+----------+------------------+
```

这样即可轻松读取 MariaDB 服务器的设置文件。通过简单查询就可以将 MariaDB 服务器中 my.cnf 设置文件的内容与 MariaDB 服务器实际拥有的全局变量（SHOW GLOBAL VARIABLES）进行比较，并掌握二者差异。

6.7.4 连接操作系统目录

通过 MariaDB 服务器的 CONNECT 存储引擎也可以访问操作系统拥有目录的子目录以及文件目录。通过下列示例进行简单讲解。

```
MariaDB> CREATE TABLE temp_dir (
           PATH varchar(256) NOT NULL flag=1,
           FNAME varchar(256) NOT NULL,
           FTYPE char(4) NOT NULL,
           SIZE double(12,0) NOT NULL flag=5)
         ENGINE=CONNECT   TABLE_TYPE=DIR   FILE_NAME='/data/*'
         OPTION_LIST='subdir=1';
```

上述代码将 TABLE_TYPE 设置为 DIR，创建要连接操作系统目录的数据表。并将 OPTION_LIST 的 subdir 选项设置为 1，表示要访问指定目录及其所有子目录。下面在 MariaDB 服务器中使用 SQL 语句访问指定目录的内容。

```
MariaDB> SELECT * FROM temp_dir;
+--------------------------------+-----------------------------+-------+--------+
| PATH                           | FNAME                       | FTYPE | SIZE   |
+--------------------------------+-----------------------------+-------+--------+
| /data/mysql/                   | multi-master                | .inf  |      0 |
| /data/mysql/performance_schema/ | performance_timers          | .frm  |   8776 |
| /data/mysql/performance_schema/ | events_waits_current        | .frm  |   9220 |
| /data/mysql/mysql/             | host                        | .MYD  |      0 |
| /data/mysql/mysql/             | time_zone                   | .frm  |   8636 |
| /data/mysql/mysql/             | host                        | .MYI  |   2048 |
...
+--------------------------------+-----------------------------+-------+--------+
```

6.8 Sequence 存储引擎 MariaDB

SEQUENCE 对象可以使 Oracle RDBMS 的序列号自增长，而 Sequence 存储引擎与该 SEQUENCE 对象完全无关。MariaDB 的 Sequence 存储引擎依据给定的条件生成序号，并创建为内存表。请绝对不要将其与 Oracle RDBMS 的序列号以及 MariaDB 的 AutoIncrement 混淆。

下面通过几个简单的示例学习 MariaDB 的 Sequence 存储引擎。首先，MariaDB 的某个特定版本可能没有安装 Sequence 存储引擎。使用 SHOW ENGINES 命令查看是否安装了 Sequence 存储引擎，若尚未安装，则使用 INSTALL PLUGIN 命令进行安装。

```
MariaDB> INSTALL PLUGIN sequence SONAME 'ha_sequence.so';
```

6.8.1 Sequence 存储引擎的基本用法

与其他存储引擎不同，Sequence 存储引擎不需要另外创建数据表。Sequence 存储引擎本身就是拥有一个数值型数据列的虚拟数据表。并且，对 Sequence 存储引擎而言，FROM 子句中指明的数据表名必须具有特定格式。Sequence 存储引擎的数据表名有如下两种格式。

```
seq_<起始值>_to_<终止值>
seq_<起始值>_to_<终止值>_step_<要跳过的值的个数>
```

第一个示例是带有记录的虚拟表，并从 <起始值> 增加（每次增 1）或减少（每次减 1）到 <终止值>。明确指出 step 时，将不按照 1 进行增减，而是以 <要跳过的值的个数> 进行增减。请看下列几个示例。

```
MariaDB> SELECT * FROM seq_1_to_5;
+-----+
| seq |
+-----+
|   1 |
|   2 |
|   3 |
|   4 |
|   5 |
+-----+

MariaDB> SELECT  * FROM seq_5_to_1;
+-----+
| seq |
+-----+
|   5 |
|   4 |
|   3 |
|   2 |
|   1 |
+-----+

MariaDB> SELECT * FROM seq_1_to_5_step_2;
+-----+
| seq |
+-----+
|   1 |
|   3 |
|   5 |
+-----+

MariaDB> SELECT * FROM seq_5_to_1_step_2;
+-----+
| seq |
+-----+
|   5 |
|   3 |
|   1 |
+-----+
```

通过上述4个示例可以很容易地了解Sequence存储引擎的使用方法。Sequence存储引擎绝对不可能产生负值，也不可能产生在特定范围内数值循环形式的结果。这是由于Sequence存储

引擎创建的值的类型为 UNSIGNED 类型。即使在 Sequence 存储引擎数据表名中强制加入“-”符号，运行时也不会产生负值。需要产生负值时，只要在 SELECT 子句中添加变换为负值的运算即可。当然，此时要先使用 CONVERT() 函数将 seq 列由 UNSIGNED INTEGER 变为 SIGNED INTEGER，然后再变换为负数。

```
MariaDB> SELECT * FROM 'seq_-5_to_1' limit 4;
+---------------------+
| seq                 |
+---------------------+
| 18446744073709551611 |
| 18446744073709551610 |
| 18446744073709551609 |
| 18446744073709551608 |
+---------------------+

MariaDB> SELECT -1 * CONVERT(seq, SIGNED INTEGER) as seq FROM seq_5_to_1;
+-----+
| seq |
+-----+
|  -5 |
|  -4 |
|  -3 |
|  -2 |
|  -1 |
+-----+
```

下面介绍几个使用 Sequence 存储引擎的示例。

6.8.2 查找遗漏的编号

假设有一个存储序列号的数据表，根据业务需求，该表必须按顺序保存编号，且不能发生遗漏。此情形下，若不编写简单的程序或用肉眼逐个查看，则很难发现遗漏的号码。

```
MariaDB> CREATE TABLE seq_test (seq INT) ENGINE=InnoDB;

MariaDB> INSERT INTO seq_test VALUES (1), (2), (3), (4), (5), (8), (9), (10);
```

此时，使用 Sequence 存储引擎进行 1 次 LEFT OUTER 连接即可轻松找到遗漏的编号。由于不需要通过读取磁盘获取数据，所以即使想要查看的数据表拥有许多记录，Sequence 存储引擎性能也不会降低。

```
MariaDB> SELECT s.seq
         FROM seq_1_to_10 s
           LEFT OUTER JOIN seq_test t ON t.seq=s.seq
```

```
        WHERE t.seq IS NULL;
+-----+
| seq |
+-----+
|   6 |
|   7 |
+-----+
```

6.8.3 创建顺序组合的号码对

实际业务中，有时需要将2个数字列组合形成编组。此时可以连接2个Sequence存储引擎数据表，轻松创建顺序组合的号码对。

```
MariaDB> SELECT s1.seq, s2.seq FROM seq_1_to_2 s1, seq_1_to_3 s2 ORDER BY s1.seq, s2.seq;
+-----+-----+
| seq | seq |
+-----+-----+
|   1 |   1 |
|   1 |   2 |
|   1 |   3 |
|   2 |   1 |
|   2 |   2 |
|   2 |   3 |
+-----+-----+
```

6.8.4 查找倍数或公倍数

查找某个指定数字的倍数或两个数字的公倍数时，连接2个Sequence存储引擎数据表即可轻松实现。

```
## 查找小于100的3的倍数
MariaDB> SELECT seq FROM seq_3_to_100_step_3;
+-----+
| seq |
+-----+
|   3 |
|   6 |
|   9 |
|  12 |
|  15 |
|  18 |
...
```

```
## 查找小于100的3与5的公倍数
MariaDB> SELECT s1.seq
        FROM seq_5_to_100_step_5 s1, seq_3_to_100_step_3 s2 ON s1.seq = s2.seq;
+-----+
| seq |
+-----+
|  15 |
|  30 |
|  45 |
|  60 |
|  75 |
|  90 |
+-----+
```

6.8.5 顺序生成字母表

利用顺序增加的数值也可以生成顺序增加的字符或字符串。

```
MariaDB> SELECT CHAR(seq) as seq_char
        FROM (
          -- 小写字母表
          (SELECT seq FROM seq_97_to_122 l)
          UNION
          -- 大写字母表
          (SELECT seq FROM seq_65_to_90 u)
          UNION
          -- 数字
          (SELECT seq FROM seq_48_to_57 d)
        ) seq_ch;

+----------+
| seq_char |
+----------+
| a        |
| b        |
| c        |
...
| A        |
| B        |
| C        |
...
| 0        |
```

```
| 1          |
| 2          |
| 3          |
...
+----------+
```

先将上述示例的结果创建两次，然后对两次结果求笛卡儿乘积，这样就能创建顺序增加的2个字的字母表字符串。

6.8.6 顺序生成日期

与生成字符串一样，使用Sequence存储引擎也可以顺序生成日期。

```
MariaDB> SELECT DATE_ADD('2014-01-29', INTERVAL s.seq-1 DAY) as seq_dat
         FROM seq_1_to_30 s;

+------------+
| seq_date   |
+------------+
| 2014-01-29 |
| 2014-01-30 |
| 2014-01-31 |
| 2014-02-01 |
| 2014-02-02 |
...
```

利用上述顺序生成日期的方法也可以按顺序增加时分秒，并生成记录。再进一步想，使用Sequence存储引擎也可以实现时间有规律的增长，比如以30分钟为单位增加时间等。

6.8.7 数据复制加工

也许各位听说过名为copy_t的数据表。定义copy_t数据表时，使之只有1个整数，该整数取值为1~*N*。该数据表常用于与其他数据表连接，以2倍或3倍复制并加工记录。像这样，只拥有1个标量值且用于复制的数据表又称互斥数据表（Mutex table）。MariaDB的Sequence存储引擎也可以用作互斥数据表。

假如要从employees数据表中访问所有工号在10001~10020的员工，从中找出最早入职员工的入职日期，并以单独记录显示。通常的做法是，先读取1次employees数据表，查找所有满足条件的员工列表，然后再另外读取1次employees数据表，查找到最早入职的员工。若使用Sequence存储引擎处理该查询，则先查询所有工号在10001~10020的员工，然后与Sequence存储引擎连接，获取所有记录副本。再根据seq值对副本进行条件分组处理，最后得到需要的结果。

```
SELECT CASE WHEN seq=1 THEN e.emp_no ELSE 'MIN_HIREDATE' END emp_no,
```

```
       CASE WHEN seq=1 THEN e.first_name ELSE '' END first_name,
       CASE WHEN seq=1 THEN e.hire_date ELSE MIN(e.hire_date) END hire_date
FROM employees e, seq_1_to_2 s
WHERE e.emp_no BETWEEN 10001 AND 10020
GROUP BY CASE WHEN seq=1 THEN e.emp_no ELSE seq END;

+--------------+------------+------------+
| emp_no       | first_name | hire_date  |
+--------------+------------+------------+
| MIN_HIREDATE |            | 1985-02-18 |
| 10001        | Georgi     | 1986-06-26 |
| 10002        | Bezalel    | 1985-11-21 |
| 10003        | Parto      | 1986-08-28 |
| 10004        | Chirstian  | 1986-12-01 |
...
```

以上为 MariaDB 的 Sequence 存储引擎使用示例。其实，Sequence 存储引擎的实现技术简单易懂。它有时就像一把钥匙，通过巧妙地编写 SQL，能够帮助我们轻松地解决遇到的问题。除了可以用作互斥表之外，Sequence 存储引擎还可以为解决其他诸多问题提供良好的解决方案。

6.9 Mroonga 全文检索存储引擎 MariaDB MySQL

Mroonga（http://mroonga.org/）插件是日本 DeNA 公司开发的全文检索引擎，最初称为 Groonga，是与 MariaDB 或 MySQL 完全无关的全文检索解决方案。随着 MySQL 与 MariaDB 用户的增长，并且考虑到 MySQL 中没有针对 CJK（Chinese、Japanese、Korean）语言的全文检索引擎，在 Groonga 检索引擎基础上，为 MySQL 与 MariaDB 开发了 Mroonga 插件。使用 Mroonga 时无需另外启动 Groonga 服务器，所以此处不讲解有关 Groonga 的内容。

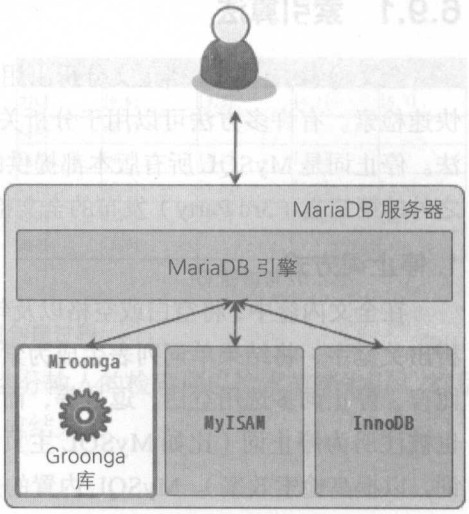


图 6-24 Mroonga 架构

由图 6-24 可知，Mroonga 是一个可以在 MariaDB 中使用的存储引擎。使用 Mroonga 时，无需在 MariaDB 外部另外启动 Groonga 服务器，只要在 MariaDB 内部使用 Groonga 库即可使用 Groonga 的全文检索功能。由于 Mroonga 存储引擎是基于数据列的存储引擎，所以 SQL 语句中使用的数据列个数越少，速度越快，性能也就越好。

Mroonga 有两种运行模式，一种是存储引擎模

式，另一种是包装器（Wrapper）模式。图 6-24 表示 Mroonga 在存储引擎模式下工作。存储引擎工作模式下，使用另外的 Groonga 存储引擎保存数据表的数据；而包装器模式下，实际数据保存于 InnoDB 或 MyISAM 存储引擎，Mroonga 只含有全文检索索引。图 6-25 表示包装器模式的构成。

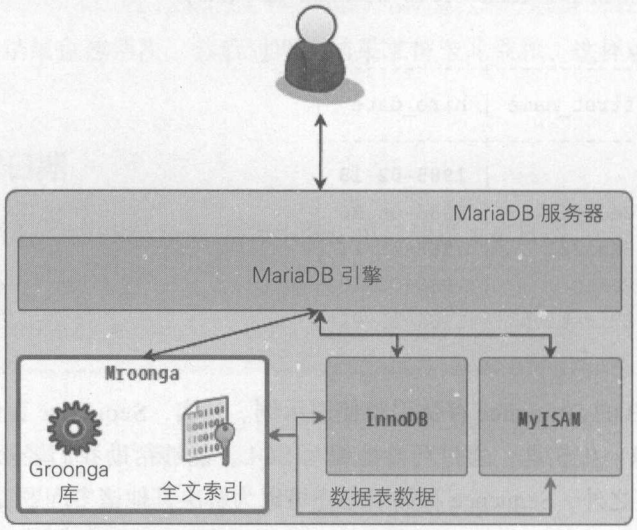


图 6-25 Mroonga 架构（包装器模式）

对于上面讲解的 Mroonga 的两种运行模式，由于 Mroonga 的具体运行模式是在创建数据表时确定的，所以编译 Mroonga 或安装 MySQL 时不必考虑运行模式，只要采用相同步骤与选项安装即可。

6.9.1 索引算法

全文检索中，从文档正文分析出用户要检索的关键字，然后构建这些关键字的索引以实现快速检索。有许多方法可以用于分析关键字与构建索引，此处只介绍停止词与 n-Gram 两种方法。停止词是 MySQL 所有版本都提供的全文检索引擎的索引方式，而 n-Gram 主要是 MySQL 之外的第三方（3rd Party）发布的全文检索引擎（Sphinx 或 Mroonga）支持的方式。

1. 停止词方式

在全文内容中，将空白或空格以及句号等句子符号[①]定义到停止词列表，然后利用停止词分析出关键字，将结果单词列表生成为索引，并在检索中使用。这种方式即为停止词方式。一般而言，停止词多选用空白、逗号等，而且特定系统中常用的单词也无检索的意义，所以大部分也被注册为停止词（比如 MySQL 主页中，单词 MySQL 就不具有检索意义，故被注册为停止词，以提高检索效率）。MySQL 内置的全文检索引擎（FullText search）只能采用这种方式建立

① 此外还包含用户定义的分隔字符串。

索引。Stopword 算法只在从文档正文抽取关键字的部分不一样，内部仍然使用 B-Tree 索引，此处不再说明。

通过全文检索引擎抽取的记录不保证按照检索词或正文内容进行排序，自然语言检索中通常会按照匹配率高低顺序进行排序显示。

2. n-Gram 方式

实际上，仅凭几个指定的停止词是不可能将全世界所有语言中的单词全部区分开来的。于是技术专家们提出了 n-Gram 方式以进行改善，该方式对无指定规则的全文也可以进行分析、检索。n-Gram 方式下，全文被几个字完全无条件地进行切分，并以此建立索引。与停止词方法相比，这种建立索引的算法更加复杂，并且创建出的索引也相当大。Mroonga 或 Sphinx 虽然也提供了其他索引建立方法，但主要使用 n-Gram 方式。n-Gram 中的“n”表示要建索引的关键字的最少字数（或字节），通常以两个字为单位切分关键字并建立索引，即经常使用 2-Gram（Bi-Gram）方式。下面主要讲解 2-Gram 方式（2 字关键字的方式）。n-Gram 索引方法通常分为前端索引与后端索引，前者管理以 2 个字为单位的最少关键字的键，后者管理 2 个字以上的关键字串（n-SubSequence Window）。建立索引的过程中，先将文档正文以大于 2 个字的大单位划分为块，以建立后端索引。然后再以 2 个字切分后端索引的关键字，形成前端索引，如图 6-26 所示。

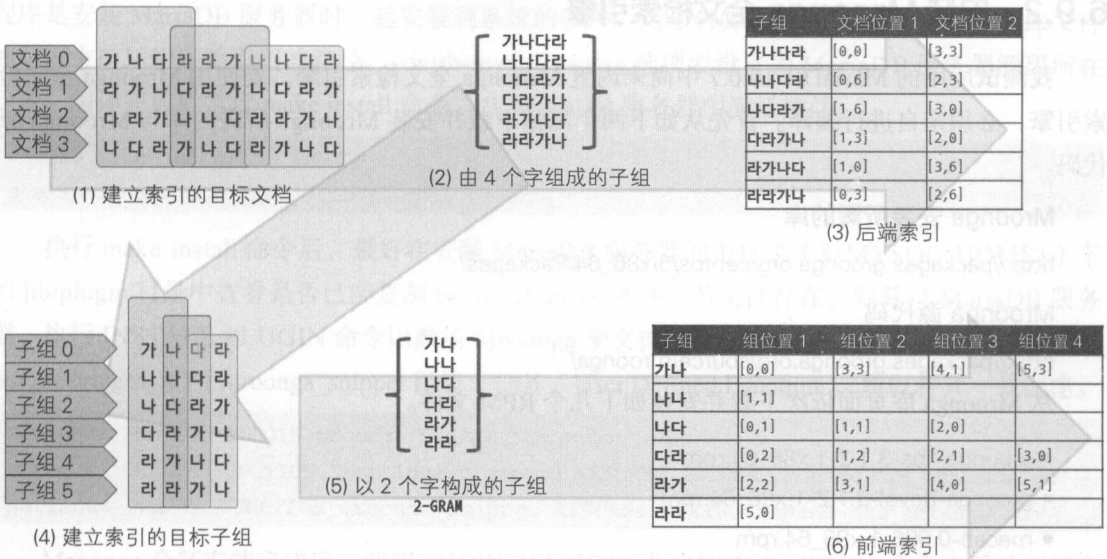


图 6-26 2-Gram 全文索引创建过程

n-Gram 方式的全文检索中，以 2 字节为单位统一切分输入的检索词以检索前端索引。之后选定候选对象，并通过后端索引进行最终验证，之后生成结果。

3. 停止词与 n-Gram 的差别

停止词方式与 n-Gram 方式最大的不同在于检索结果。下面以带有如下数据的数据表为示例进行说明。

doc_id	doc_body
1	销售二手 iPhone 3G
2	求购 iPhone 3Gs
3	贱卖 iPhone 3Gs

在以上表格的 doc_body 数据列创建全文检索索引并执行如下查询，观察结果有何不同。

```
select * FROM tb_test WHERE MATCH(doc_body)AGAINST('iPhone' IN BOOLEAN MODE);
```

停止词方式（MySQL 5.5 与 5.6 的内置全文检索）

doc_id	doc_body
1	销售二手 iPhone 3G
2	求购 iPhone 3Gs

n-Gram 方式（Mroonga 存储引擎）

doc_id	doc_body
1	销售二手 iPhone 3G
2	求购 iPhone 3Gs
3	贱卖 iPhone 3Gs

MySQL 内置的全文检索引擎使用停止词方式，在该方式检索中，为了以停止词为基准进行左一致或全体一致的检索比较，检索词（“iPhone”）前面一定不要有其他词（“苹果”），否则查找不到。

6.9.2 安装 Mroonga 全文检索引擎

我测试所用的 MariaDB 10.0.7 中尚未内置 Mroonga 全文检索引擎。要使用 Mroonga 全文检索引擎，必须亲自进行编译。首先从如下两个网站下载并安装 Mroonga 所需的库与 Mroonga 源代码。

Mroonga 安装所需的库

http://packages.groonga.org/centos/5/x86_64/Packages/

Mroonga 源代码

http://packages.groonga.org/source/mroonga/

从 Mroonga 库页面依次下载并安装如下几个 RPM 文件。

- groonga-libs-3.1.2-1.x86_64.rpm
- groonga-devel-3.1.2-1.x86_64.rpm
- mecab-0.996-1.x86_64.rpm
- mecab-ipadic-2.7.0.20070801-8.1.x86_64.rpm
- groonga-tokenizer-mecab-3.1.2-1.x86_64.rpm
- groonga-normalizer-mysql-1.0.5-1.x86_64.rpm

从 Mroonga 源代码页面下载如下两个文件之一。此时，若已经安装 MariaDB 10.0.x，则只需下载 mroonga-3.12.tar.gz；若尚未安装 MariaDB 而希望安装时也编译生成 MariaDB，则只需下载 mariadb-10.0.7-with-mroonga-3.12.tar.gz 文件。

- mariadb-10.0.7-with-mroonga-3.12.tar.gz
- mroonga-3.12.tar.gz

编译 mariadb-10.0.7-with-mroonga-3.12.tar.gz 文件与编译 MariaDB 10.0.x 的方法几乎没什么不同，此处只讲解 mroonga-3.12.tar.gz 的编译方法。安装 mroonga-3.12.tar.gz 前，先要安装 MariaDB 10.0.x，服务器中必须存在与安装版本相同的 MariaDB 源代码（MariaDB 网站发布的不含 mroonga 源代码的 MariaDB 10.0.x 的源代码）。下面解压缩并转到 mroonga 源目录，执行 configure 与 make 命令。

```
shell> mroonga-3.12.tar.gz
shell> cd mroonga-3.12

shell> ./configure \
--with-mysql-source=/root/install/mariadb-10.0.7 \
--with-mysql-config=${MYSQL_HOME}/bin/mysql_config

shell> make
```

配置（configure）Mroonga 源代码时，--with-mysql-config 选项中设置的 mysql_config 实用程序是安装 MariaDB 服务器时一起安装到系统的程序，用于告知引用 MariaDB 头文件与库文件的第三方应用程序要引用的路径。--with-mysql-source 选项要设置为 MariaDB 服务器源码所在目录。编译完成后执行 make install 命令，在 MariaDB 服务器中激活插件。

```
shell> make install
shell> ls ${MYSQL_HOME}/lib/plugin/
```

执行 make install 命令后，最好在安装 MariaDB 服务器的主目录（${MYSQL_HOME}）下的 lib/plugin 目录中查看是否已经复制 ha_mroonga.so 文件。若文件存在，则登录 MariaDB 服务器，执行 INSTALL PLUGIN 命令以激活 Mroonga 全文检索引擎。由于使用 Mroonga 时要用到 last_insert_grn_id 与 mroonga_snippet 函数（UDF，User Defined Function），所以将其一并创建。

```
MariaDB> INSTALL PLUGIN mroonga SONAME 'ha_mroonga.so';
MariaDB> CREATE FUNCTION last_insert_grn_id RETURNS INTEGER soname 'ha_mroonga.so';
MariaDB> CREATE FUNCTION mroonga_snippet RETURNS STRING soname 'ha_mroonga.so';
```

Mroonga 全部安装完成后，使用 SHOW PLUGINS 或 SHOW ENGINES 命令查看是否正常激活 Mroonga。

6.9.3 使用 Mroonga 全文检索引擎

下面创建一个使用 Mroonga 全文检索引擎的数据表。

```
MariaDB> CREATE TABLE tb_test (
    id INT PRIMARY KEY AUTO_INCREMENT,
```

```
    content VARCHAR(255),
    FULLTEXT INDEX fx_content (content)
                COMMENT 'parser "TokenBigramIgnoreBlankSplitSymbolAlphaDigit"'
) ENGINE=mroonga COMMENT='engine "innodb"' DEFAULT CHARSET utf8mb4;

MariaDB> INSERT INTO tb_test (content) VALUES
         ("今天学习了MariaDB，明天也要学习MariaDB。");
MariaDB> INSERT INTO tb_test (content) VALUES
         ("" 今天学习了MariaDB，但明天要学习MySQL。"");
```

FULLTEXT INDEX 行尾的 COMMENT 中出现的 parser "TokenBigramIgnoreBlankSplitSymbol AlphaDigit" 用于设置 Mroonga 全文检索引擎如何处理特殊字符、字母表以及数字等。在 Mroonga 中可以使用如下多种分析器选项。

OFF

Mroonga 全文检索引擎中，使用停止词[①]切分文本字符串并生成标记的过程称为“标记化”（tokenizing）。“标记”是指从字符串去除特殊字符与空格等之后的、对检索有意义的单词。比如“全文检索是什么?”字符串中，大部分用户对问号等标点符号并不感兴趣，所以将整个字符串切分为“全文”、“检索”、“是什么”三个标记，并建立检索索引。由于未执行“)”，所以存储的内容保持不变，只允许检索或前端匹配。

TokenBigram

使用 Bi-Gram 算法分解字符串。此时，连续的字母、数字、特殊字符都被认定为标记，故允许存在 3 个字以上的标记。TokenBigram 被设置为默认分析器。

TokenMecab

使用 MeCab 库分解字符串。为此，MeCab 库要随 Mroonga 一起创建生成。MeCab 库是日语专用词素分析（morphological analyzer）库。

TokenBigramSplitSymbol

使用 Bi-Gram 算法分解字符串。与 TokenBigram 不同，连续的特殊字符不被认作标记，采用 Bi-Gram 分解字符串。也就是说，若用“!?”执行全文检索，则 TokenBigramSplitSymbol 分析器选项中，与“Is it really!?!?”的“!?!?”进行匹配，并获取检索结果。但“TokenBigram”分析器选项中，“Is it really!?!?”不匹配，检索结果为空。

TokenBigramSplitSymbolAlpha

使用 Bi-Gram 算法分解字符串，与 TokenBigramSplitSymbol 一样，连续的特殊字符也不被认作标记，采用 Bi-Gram 分解字符串。比如，某条记录的列值为“Is it really?”，使用“real”进行检索时，在 TokenBigramSplitSymbolAlpha 分析器选项中能够得到匹配的结果，但在 TokenBigram 中无法得到匹配的结果。TokenBigram 分析器选项中一定要使用“really”才能得到匹配的结果。

---

① 用于指定输入字符串的停止词。（停止词可以是特殊字符、空格、句子符号等）

TokenBigramSplitSymbolAlphaDigit

使用 Bi-Gram 分解字符串，与 TokenBigramSplitSymbolAlpha 一样，连续的数字也不被认作标记，采用 Bi-Gram 分解字符串。比如，某条记录的列值为“090-0123-4567”，在 TokenBigramSplitSymbolAlphaDigit 分析器选项中，“567”检索词会得到匹配的结果；而在 TokenBigram 分析器选项中，“567”检索词无法获得任何结果值。okenBigram 分析器选项中必须用“4567”进行检索才能获得结果。

TokenBigramIgnoreBlank、TokenBigramIgnoreBlankSplitSymbol、TokenBigramIgnoreBlankSplitSymbolAlpha、TokenBigramIgnoreBlankSplitSymbolAlphaDigit

这 4 种分析器选项全部采用 Bi-Gram 分解字符串，执行分析时，分别与 TokenBigram、TokenBigramSplitSymbol、TokenBigramSplitSymbolAlpha、TokenBigramSplitSymbol AlphaDigit 相似。但在 IgnoreBlank 模式的分析器选项中，所有空格都被忽略。也就是说，TokenBigram 中的空格被认作无效字符，而 TokenBigramIgnoreBlank 分析器选项中的所有空格都会被忽略。因此，“大家好”与“大 家 好”被视为相同字符串。

此外还提供了其他几种分析器选项，由于不太重要，故不再详解。更多相关内容请浏览分析器选项说明页面（http://mroonga.org/docs/tutorial/storage.html）。

本书示例数据表中，所有空格字符都被忽略。为了对字母、数字、特殊字符进行前缀匹配（Prefix match）及中间部分匹配检索，将分析器选项设置为 TokenBigramIgnoreBlankSplitSymbolAlphaDigit。

下面对几种全文检索查询进行测试。

```
MariaDB> SELECT *, MATCH (content) AGAINST ('MariaDB' IN BOOLEAN MODE) as score
         FROM tb_test
         WHERE MATCH (content) AGAINST ('MariaDB' IN BOOLEAN MODE)
         ORDER BY MATCH (content) AGAINST ('MariaDB' IN BOOLEAN MODE) DESC;

+----+-----------------------------------------------------+-------+
| id | content                                             | score |
+----+-----------------------------------------------------+-------+
| 1  | 今天学习了 MariaDB，明天也要学习 MariaDB。          |     2 |
| 2  | 今天学习了 MariaDB，但明天要学习 MySQL。            |     1 |
+----+-----------------------------------------------------+-------+

MariaDB> SELECT *, MATCH (content) AGAINST ('ariaDB' IN BOOLEAN MODE) as score
         FROM tb_test
         WHERE MATCH (content) AGAINST ('ariaDB' IN BOOLEAN MODE)
         ORDER BY MATCH (content) AGAINST ('ariaDB' IN BOOLEAN MODE) DESC;

+----+-----------------------------------------------------+-------+
| id | content                                             | score |
+----+-----------------------------------------------------+-------+
```

```
| 1  | 今天学习了MariaDB，明天也要学习MariaDB。                    |     2 |
| 2  | 今天学习了MariaDB，但明天要学习MySQL。                      |     1 |
+----+-------------------------------------------------------+-------+
```

注意 本书所用的MariaDB版本（10.0.8）中，Mroonga全文检索引擎尚未被提供为MariaDB存储引擎。若想使用，要先下载MariaDB源码与Mroonga源码，然后对其进行编译，从而得到Mroonga插件库。Monty Program AB公司正在进行讨论（当前目标是将Mroonga存储引擎包含到MariaDB 10.0.9），将Mroonga全文检索引擎提供为MariaDB 10.0的默认存储引擎。鉴于Oracle MySQL的全文检索引擎并未添加基于n-Gram的分析算法，Mroonga可能是目前MySQL与MariaDB中针对全文检索的唯一解决方案。

第7章

其他功能

7.1 性能提升 MariaDB MySQL Enterprise

7.2 管理与诊断 MariaDB

7.3 开发效率 MariaDB

7.4 分区 MariaDB MySQL

7.5 备份 MariaDB MySQL

MariaDB 与 MySQL 服务器并不总发布相同功能。MySQL 服务器本身有付费版本——MySQL 企业版，所以它不能原封不动地使用 MariaDB 或 PerconaServer 的新功能。而 MariaDB 本身无付费版本，所有版本都遵循 GPL 协议进行发布，所以可以随时使用 MySQL 社区版或 PerconaServer 的优秀功能。像这样，MariaDB 可以不受限制地使用不同版本 MySQL 服务器中的优秀功能。但仍然要解决相互兼容以及一些技术问题，才能将其他版本的优秀功能添加到 MariaDB。

基于以上原因，本书也全面讲解了有关 MySQL 或 PerconaServer 的功能。有些功能即使目前尚未应用到 MariaDB，但肯定很快就会内置到 MariaDB。MariaDB 虽然是从 MySQL 服务器派生出的一个分支（Fork）版本，但绝对不是 MySQL 服务器所有功能的部分集合。它在 MySQL 功能的基础上添加了新功能或对原有功能进行了改善，所以是一个超集版本。本章将介绍 MariaDB 10.0.x 版本拥有的新功能，这些功能有些是 MySQL 5.6 版本的新功能，也有些是 MariaDB 自身具有的功能。

7.1 性能提升 MariaDB MySQL Enterprise

MariaDB 版本升级到 10.0 后，优化器、各种基本处理算法以及 InnoDB 或 XtraDB 自身性能也有了很大提升。这些内容前面已经讲过，下面只介绍 MariaDB 的线程池（Thread Pool），它是 MariaDB 重要的功能之一。

线程池

MySQL 5.5 或 MySQL 5.6 社区版中不能使用线程池，要使用线程池功能，则必须使用 Oracle MySQL 企业版。各位可能经常看到图 7-1 所示的性能曲线，每当谈论 MySQL 5.5 或 MySQL 5.6 的性能或扩展性提升时，常常使用图 7-1 中的性能比较曲线。同时到来的客户端连接或请求增多时，会一直保持峰值级别的处理量，就像曲线右侧标注的一样，但这仅仅针对 MySQL 企业版。令人遗憾的是，免费的 MySQL 社区版中仍然不能使用线程池功能。

1. MySQL 服务器传统的连接及其处理方式

传统方式下，每当有 1 个客户端连接 MySQL 服务器时，MySQL 服务器就会为该客户端创建 1 个单独的线程以提供服务。也就是说，若有 10 000 个客户端连接服务器，MySQL 服务器必须创建 10 000 个线程以响应用户请求。然而，线程数越多，对服务器软硬件资源的竞争就越激烈，性能自然会下降。图 7-2 表示 MySQL 服务器创建线程的传统方式。从图 7-2 中可以看到，无论线程处于空闲状态（Idle）还是运行状态（Running），都要根据所连接的用户数创建线程。

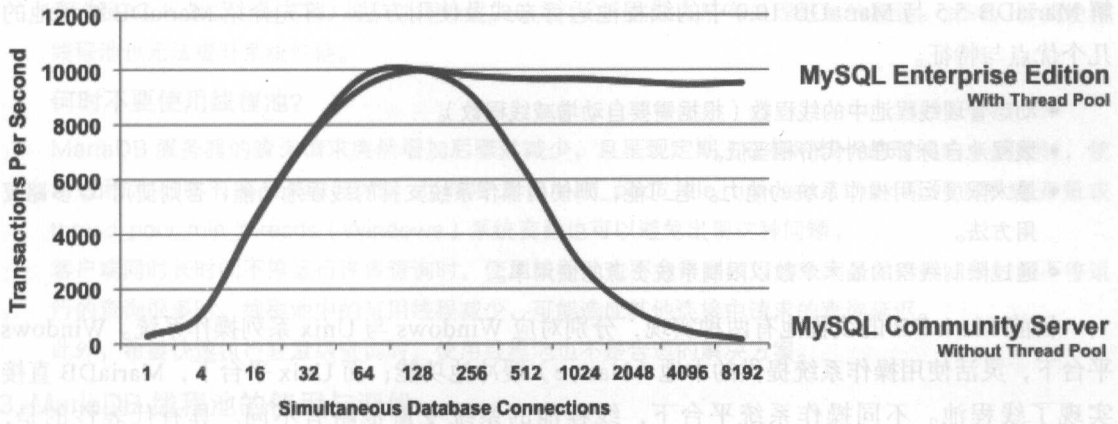


图 7-1 MySQL 线程池性能

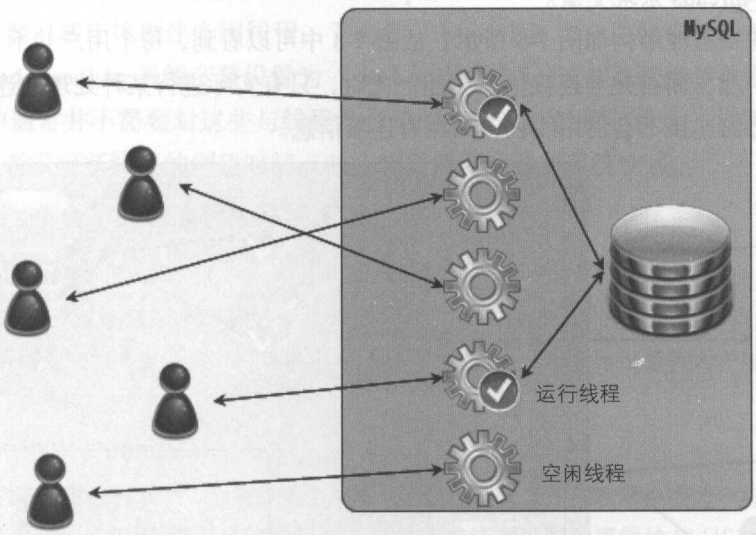


图 7-2 MySQL 服务器创建线程的传统方式

大部分线程不进行任何操作，即处于空闲状态时，不会对整体性能产生太大影响。但同时执行的线程增加时，操作系统的上下文就会频繁切换。这种上下文切换本身会占用系统开销，降低 CPU 缓存局部性（Cache Locality）。若许多线程同时竞争相同资源，情况就更加恶化。为解决该问题而引入线程池机制，减少同时运行的线程（Concurrent thread）数，降低上下文切换次数。即便如此，无条件减少线程个数并非最佳解决方案。最好的解决方法是每个核心维持 1 个激活的线程，但这在现实世界中很难实现。

2. MariaDB 的线程池

MariaDB 5.1 开始实现线程池功能，并不断改善，一直到 MariaDB 10.0。MariaDB 的线程池由 Monty Program AB 公司实现，现在由 Monty Program AB 与 Percona 公司共同改进。下面将讲

解 MariaDB 5.5 与 MariaDB 10.0 中的线程池运行方式及使用方法。首先介绍 MariaDB 线程池的几个优点与特征。

- 动态管理线程池中的线程数（根据需要自动增减线程数）。
- 线程池自身管理的代价相当低。
- 最大限度运用操作系统的能力。若可能，则使用操作系统支持的线程池功能，否则使用 IO 多路复用方法。
- 通过限制线程的最大个数以限制系统资源的使用率。

目前 MariaDB 的线程池有两种实现，分别对应 Windows 与 Unix 系列操作系统。Windows 平台下，灵活使用操作系统提供的本地（Native）缓冲池功能；而 Unix 平台下，MariaDB 直接实现了线程池。不同操作系统平台下，线程池的系统变量也略有不同。具有代表性的是，Windows 平台不支持 thread_pool_size 系统变量，而 Unix 系列平台的线程池形式上只支持 threadpool_min_threads 系统变量。

MariaDB 的线程池结构如图 7-3 所示。从图 7-3 中可以看到，每个用户并不直接与 MariaDB 的线程连接在一起，而首先与连接信息相连。此时，只有实际进行某种处理的连接才占用线程，而处于空闲状态的连接不会占用线程，只拥有连接信息。

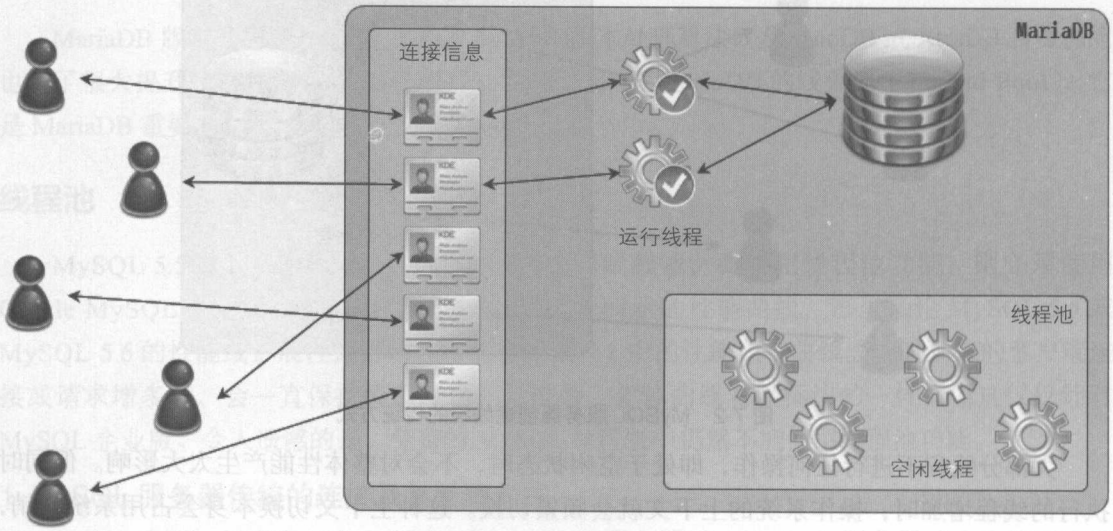


图 7-3 MariaDB 的线程池结构

某个特定用户执行查询时，MariaDB 服务器就会查看线程池中有无空闲线程，然后将可用线程分配给相应连接（用户）。若线程池中无可用线程，线程池就会创建新线程并分配给用户连接。线程池中空闲线程太多时，就会按照一定规则删除。该过程中会有多种选择事项，这些选择可以通过 MariaDB 服务器中与线程池相关的系统变量进行控制。

何时使用线程池最有效率？

一般而言，快速执行每个查询的 CPU 密集型作业中，使用线程池会非常有效率。若 MariaDB 执行的

查询是 IO 密集型，则 CPU 等待处理较慢的磁盘 I/O，这成为所有处理性能的瓶颈。此时，即使使用线程池也无法提升系统性能。

何时不要使用线程池?

MariaDB 服务器的查询请求突然增加后骤然减少，且呈现定期（在相对较长的时间内）反复时，使用线程池不会有高效率。当然，此时增大 thread_pool_idle_timeout（Unix 系列）系统变量或 thread_pool_min_threads（Windows）系统变量也可以避免出现这种问题。

客户端同时长时间不停运行许多查询时，使用线程池也不会得到较好的效率。像这样，长时间不停运行的查询很多时，线程池中的可用线程减少，可能造成其他连接中请求的查询延迟。

此外，希望快速执行轻量级查询时，使用线程池也不是合适的解决方案。

3. MariaDB 线程池的使用与调优

若想在 MariaDB 中使用线程池，必须先将 thread_handling 系统变量设置为“pool-of-threads”。thread_handling 的默认值为“one-thread-per-connection”，这是引入线程池之前的工作方式，即为每个客户端连接创建专用线程。下面学习 MariaDB 中有关线程池的系统变量。请注意，只有将 thread_handling 系统变量设置为“pool-of-threads”，这些系统变量才起作用。

MariaDB 中通常并不需要对这些与线程池有关的变量进行设置与优化。但需要设置线程池的最少线程数或者设定新线程的创建时间点时，就需要调整这些系统变量。

(1) Windows 平台下线程池相关系统变量

thread_pool_min_threads

该系统变量用于设置线程池中要拥有的最少线程数，默认值为 1。长时间没有用户查询请求进入 MariaDB 服务器而突然大量涌入时，可以将 thread_pool_min_threads 设置得大一些，增加线程池中默认要创建的线程数。

thread_pool_max_threads

该系统变量用于设置线程池中可拥有的最大线程数，默认值为 500，表示线程池中所能拥有的最大线程数不超过 500 个。同时需要很多线程时（系统故障或执行需要全局锁的 FLUSH TABLES WITH READ LOCK 等查询时），许多客户查询不得不等待。这样，所需线程数可能超过线程池允许的最大线程数。针对该情形，可以将 thread_pool_max_threads 系统变量的值设置得大一些。有时也需要将线程池可拥有的最大线程数设置得比默认值小，这样可以限制 MariaDB 使用的内存量，或者保证整体处理量。

(2) Unix 系列线程池相关系统变量

thread_pool_size

该系统变量用于设置线程池中的线程组个数，默认为服务器中处理器（核心）的个数，也是同时进行作业处理并获得最佳性能的个数。也就是说，一个系统中，进程（Process）的启动数与处理器（Processor）数一致，上下文切换次数最少且 CPU 的缓存局部性最好时，才能获得最佳性能。Unix 系列的缓冲池中，所有客户连接被划分为多个不同组。MariaDB 在一个时间点从每个组只调用 1 个线程执行。有时，MariaDB 以外的进程可以分配使用专用的 CPU 核心，此时，最好将 thread_pool_

size 值设置得比核心数少。MariaDB 5.5 支持该系统变量，但 MariaDB 10.0 不再支持。

thread_pool_stall_limit

线程池的主要作用在于，维持最少数量的线程，将对系统或软件资源的竞争降到最低，从而使 MariaDB 的处理量最大化。线程池中再无可用线程时，thread_pool_stall_limit 系统变量决定等待多少时间后创建新线程，单位为毫秒。在合适范围内（其值设置过大则负面影响越大），其值越大，MariaDB 服务器的整体处理量就越高。但需要注意，thread_pool_stall_limit 系统变量的值越大，线程就要等待相应更长的时间，特定用户的查询延迟就会非常明显。相反，thread_pool_stall_limit 系统变量值越小，用户之间响应时间的不均衡虽然会减少，但同时会降低线程池的效能。起初先将该系统变量设置为 300~500 毫秒，然后检查客户端的响应时间是否满足 SLA（Service-level-agreement），然后再逐渐调低，这种调优方式是最好的。MariaDB 5.5 支持该系统变量，但 MariaDB 10.0 不再支持。

thread_pool_max_threads

请参考前文有关 thread_pool_max_threads 系统变量的说明。Unix 系列操作系统中，正在启动的 MariaDB 内，该变量的默认值为 500。

thread_pool_idle_timeout

线程池中处于空闲状态的线程数大于最少线程数时，线程的个数就会自动减少。此时并非直接删除过剩线程，而要先等待 thread_pool_idle_timeout 时间，然后删除线程。若服务请求定期暴增又减少，且该过程不断反复，最好将该变量值设置得大一些，以使线程池中的线程不会立即删除。MariaDB 5.5 支持该系统变量，但 MariaDB 10.0 不再支持。

thread_pool_oversubscribe

该系统变量为内部参数，若可能，请尽量不要调整。更改内部参数后，可能不会出现用户手册中记述的反应或结果。MariaDB 5.5 支持该系统变量，但 MariaDB 10.0 不再支持。

线程池中的线程数超过 thread_pool_max_threads 系统变量的设置值时，客户将无法再连接服务器。MySQL 服务器传统的线程创建方式“one-thread-per-connection”中也存在类似情形，即 max_connections 系统变量中设置的连接数全部用光时，用户将无法再连接服务器。使用线程池时，出现上述情形后，可以将连接连到其余端口。下面两个系统变量用于设置另外的连接通道，使线程池充满时仍可连接 MariaDB 服务器，以调整线程池有关设置。

extra_port

该系统变量用于设置 MariaDB 服务端口之外的端口，该端口供管理员管理服务器，要与 MariaDB 服务器对外提供服务的端口不同。

extra_max_connections

该系统变量用于设置 extra_port 端口允许的最大连接数。extra_port 与 extra_max_connections 系统变量创建管理通道，采用“一连接一线程”的方式。有 10 个连接连至 extra_port 时，MariaDB 服务器就会相应创建 10 个线程。

除了这些用于控制线程池工作方式的系统变量外，MariaDB 还提供了下列两个状态变量，用于查看线程池的工作状态。检测这两个状态变量可以知道目前 MariaDB 服务器中平均正在使

用几个线程，以及有几个线程处于空闲状态。借助这些信息，我们可以对线程池的各个系统变量再次进行调优。

Threadpool_threads

线程池中的线程个数。

Threadpool_idle_threads

线程池的线程中尚未处理客户端查询请求且处于空闲状态的线程数。

4. 注意事项

线程池激活时，MariaDB 的线程缓存（由 thread_cache_size 系统变量决定）自动进入禁用状态。因此，使用线程池时无需再监控 MariaDB 的线程缓存状态。

另外，线程池充满时将无法正常工作。此时，MariaDB 会在错误日志中显示如下信息。定期查看错误日志文件可知，thread_pool_max_threads 系统变量所设数量的线程是否已经全部耗尽而无法再创建线程。

```
120721 17:59:47 [ERROR] Threadpool could not create additional thread to handle
queries, because the number of allowed threads was reached. Increasing 'thread_pool_
max_threads' parameter can help in this situation.
If ‹extra_port› parameter is set, you can still connect to the database with
superuser account (it must be TCP connection using extra_port as TCP port) and
troubleshoot the situation. A likely cause of pool blocks are clients that lock
resources for long time. ‹show processlist' or 'show engine innodb status' can
give additional hints.
120721 17:59:47 [Note] Threadpool has been blocked for 30 seconds
```

7.2 管理与诊断 MariaDB

MySQL 与 MariaDB 服务器中，用于诊断服务器状态的功能仍然十分不足。当然，MySQL 5.6 开始进行了改进，管理员通过 PERFORMANCE_SCHEMA 可以诊断多种锁或等待状态，但用于诊断内存或各客户端线程的状态等功能还远远不够。MariaDB 10.0 对这些部分逐步进行改善，添加了诸多汇报功能。借助这些功能，管理员可以轻松获得特定线程执行查询的执行计划以及各线程的内存使用量。

7.2.1 SHOW EXPLAIN FOR<THREAD-ID>

对于正在处理特定客户端请求的线程，从 MariaDB 10.0 开始，可以直接查看其正在运行的查询的执行计划。下面是 MariaDB 服务器运行两个 mysql 客户端的结果，其中一个执行访问 employees 数据表的查询，另一个连接访问进程列表。从运行结果可知，3 号线程正在查询（SELECT）employees 数据表。

```
MariaDB> SHOW PROCESSLIST;
+----+------+---------+-------------------------------------------------------------------+
| Id | User | Command | Info                                                              |
+----+------+---------+-------------------------------------------------------------------+
|  3 | root | Query   | SELECT * FROM employees WHERE emp_no BETWEEN 10001 AND 10030       |
|  4 | root | Query   | show processlist                                                  |
+----+------+---------+-------------------------------------------------------------------+
```

访问进程列表的连接中，使用“SHOW EXPLAIN FOR”命令访问3号线程正在运行的查询的执行计划，如下所示。

```
MariaDB> SHOW EXPLAIN FOR 3;
+------+-------------+-----------+-------+---------+---------+------+------+-------------+
| id   | select_type | table     | type  | key     | key_len | ref  | rows | Extra       |
+------+-------------+-----------+-------+---------+---------+------+------+-------------+
|    1 | SIMPLE      | employees | range | PRIMARY | 4       | NULL |   30 | Using where |
+------+-------------+-----------+-------+---------+---------+------+------+-------------+
1 row in set, 1 warning (0.02 sec)
```

当然，也可以复制3号线程正在执行的查询，使用EXPLAIN命令查看执行计划。但是，众所周知，使用EXPLAIN命令查看的查询执行计划是优化器利用查询的结构与统计信息模拟查询执行而得到的。也就是说，实际执行查询时，并非一定会按照EXPLAIN显示的执行计划执行。而使用SHOW EXPLAIN命令显示的执行计划是当前相关线程制定并正在执行的执行计划。因此，使用SHOW EXPLAIN命令可以查看查询实际执行时采用的执行计划。

到MySQL 5.5版本之前，统计信息都不是静态不变的。触发一定条件时就会重新收集统计信息，同时，原来一直使用索引的查询也会经常突然发生全表扫描的情形。然而，此时若使用EXPLAIN命令查看该查询的执行计划，则不会发现任何问题，看上去查询仍然在使用索引。针对该问题，MariaDB 10.0开始提供SHOW EXPLAIN命令，使用该命令可以查看特定线程实际采用的执行计划。执行SHOW EXPLAIN命令后，总会出现1个警告信息（1 warning）。使用SHOW WARNINGS命令查看警告信息，可以查看目标线程正在执行的SQL语句。

有时，执行SHOW EXPLAIN命令会失败。只有查询需要执行计划且线程正在执行这样的查询时，才能使用SHOW EXPLAIN命令，否则会发生如下错误。

```
MariaDB> SHOW EXPLAIN FOR 3;
ERROR 1933 (HY000): Target is not running an EXPLAINable command
```

此外，MySQL与MariaDB服务器中也可以对查询的一部分推迟优化，立即删除不再需要的部分的执行计划。因此，SHOW EXPLAIN命令执行结果的Extra字段中，会出现Not yet optimized或Query plan already deleted语句。

7.2.2 向慢查询日志输出执行计划

从 MariaDB 10.0.5 开始，不仅可以向慢查询日志输出查询的执行时间及所执行的查询，还可以一并输出查询所用的执行计划。MariaDB 5.5 开始提供 log_slow_verbosity 系统变量，该系统变量可以设置为如下几个值。

microtime

以微秒为单位，显示慢查询日志的各种时间相关信息。

query_plan

输出 SELECT 查询执行计划的简单信息。将 log_slow_verbosity 系统变量设置为 query_plan 时，向慢查询日志文件输出 Full_scan 或 Full_join、Tmp_table 与 Tmp_table_on_disk、Filesort 与 Filesort_on_disk，以及 Merge_passes during sorting 等内容。

full

设置 log_slow_verbosity 系统变量时，可以混合使用多个选项。此时可以将其设置为 full，获得类似同时使用所有选项的效果。

explain

从 MariaDB 10.5 开始添加的选项。将 log_slow_verbosity 系统变量设置为 explain 后，MariaDB 服务器会将查询的 EXPLAIN 结果值全部记录到慢查询日志。

注意 除 log_slow_verbosity 系统变量外，MariaDB 服务器还提供了 log_slow_filter 与 log_slow _rate_limit 两个系统变量。MariaDB 执行的查询的执行计划中，log_slow_filter 系统变量将满足特定条件的查询记录到慢查询日志。log_slow_filter 系统变量可以设置为如下这些值。

- admin：将管理员使用的查询（CREATE、ALTER、OPTIMIZER、DROP 等）全部记录到慢查询日志。
- filesort：将所有伴有排序的处理的查询记录到慢查询日志。
- filesort_on_disk：将使用磁盘而非内存进行排序处理的查询记录到慢查询日志。
- full_join：将不能使用索引的连接记录到慢查询日志。
- full_scan：将执行全表扫描的查询记录到慢查询日志。
- query_cache：将采用查询缓存处理的查询记录到慢查询日志。
- query_cache_miss：将无法采用查询缓存处理的查询记录到慢查询日志。
- tmp_table：将使用临时表的查询记录到慢查询日志。
- tmp_table_on_disk：将在磁盘上创建临时表的查询记录到慢查询日志。

一次性向慢查询日志写入太多内容时，MariaDB 服务器就会为慢查询日志消耗过多系统资源。为了防止出现这一情形，MariaDB 服务器中的 log_slow_rate_limit 系统变量将不进行日志记录的查询设置为合适的个数，避免同时发生太多慢查询日志。

将 log_slow_verbosity 系统变量设置为 explain 后，即可同时查看查询的详细执行计划，如下所示。下列示例将 log_slow_verbosity 系统变量设置为“query_plan,explain”之后，执行慢查询。因此，在下列示例中可以查看类似“Full_scan:Yes”的执行计划概要，也可以查看查询执行计划的所有详细内容。

```
# Time: 140201 22:47:15
# User@Host: root[root] @ localhost [127.0.0.1]
# Thread_id: 3  Schema: employees  QC_hit: No
# Query_time: 2.199310  Lock_time: 0.000148  Rows_sent: 0  Rows_examined: 663206
# Full_scan: Yes  Full_join: No  Tmp_table: No  Tmp_table_on_disk: No
# Filesort: No  Filesort_on_disk: No  Merge_passes: 0
#
# explain: id   select_type   table   type     possible_keys       key       key_len  ref
rows Extra
# explain: 1    SIMPLE   de   ALL     ix_empno_fromdate NULL       NULL      N  U  L  L
331603
# explain: 1    SIMPLE   e    eq_ref  PRIMARY   PRIMARY  4          employees.de.emp_no
1   Using where
#
use employees;
SET timestamp=1391262435;
SELECT * FROM employees e, dept_emp de WHERE de.emp_no=e.emp_no AND e.last_name='Matt';
```

7.2.3 输出结构化的执行计划

从MySQL 5.6开始，查询的执行计划可以采用JSON格式进行输出。应用程序之前可以通过EXPLAIN命令直接取走查询的执行计划，所以是否采用JSON就显得不太重要。MySQL 5.6开始添加的JSON格式中，原本用EXPLAIN命令无法查看的详细内容也会一起记录下来，这是非常重要的。请注意，MariaDB 10.0.7尚不支持该功能。

在MySQL 5.6中使用EXPLAIN FORMAT=JSON命令查看简单查询的执行计划。

```
mysql> EXPLAIN FORMAT=JSON
       SELECT COUNT(*) FROM employees e, dept_emp de WHERE de.emp_no=e.
emp_no AND e.last_name LIKE '%M%';

EXPLAIN: {
  "query_block": {
    "select_id": 1,
    "nested_loop": [
      {
        "table": {
          "table_name": "e",
          "access_type": "ALL",
          "possible_keys": [
            "PRIMARY"
          ],
          "rows": 299556,
```

```
        "filtered": 100,
        "attached_condition": "('employees'.'e'.'last_name' like '%M%')"
      }
    },
    {
      "table": {
        "table_name": "de",
        "access_type": "ref",
        "possible_keys": [
          "ix_empno_fromdate"
        ],
        "key": "ix_empno_fromdate",
        "used_key_parts": [
          "emp_no"
        ],
        "key_length": "4",
        "ref": [
          "employees.e.emp_no"
        ],
        "rows": 1,
        "filtered": 100,
        "using_index": true
      }
    }
  ]
 }
}
```

如上述示例结果所示，仅使用 EXPLAIN 命令也可以查看很多信息。但上述结果含有“attached_condition”部分，通过它可以查看 MySQL 服务器的各处理部分使用 WHERE 语句的哪些条件进行执行。此外，MySQL Workbench 工具中，借助 JSON 格式的执行计划可以将 EXPLAIN 结果以可视化方式显示。图 7-4 在 MySQL Workbench 中以 Visual EXPLAIN 方式显示上述查询的执行计划。

如图 7-4 所示，MySQL Workbench 的 Visual Explain 中，红方框显示全表扫描，绿方框显示索引查找，能够轻松把握哪些部分会对性能产生影响。

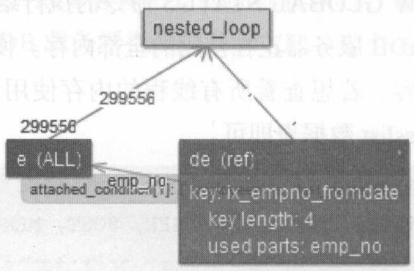


图 7-4 MySQL Workbench 的 Visual Explain

7.2.4 线程的内存使用量

针对处理服务器或各客户端请求的线程，MySQL 服务器提供的用于诊断其内存使用状态的功能还远远不够。MySQL 服务器使用的内存为 10 GB，除了缓冲池、查询缓存以及 MyISAM 的主缓存内存之外，完全没有其他告知信息。对于这些部分，人们总是根据使用手册中的内容、仅通过预测判断内存的使用量。MariaDB 10.0 添加了相应功能，用户借助该功能可以查看处理各客户端请求的线程正在使用的内存。

MySQL 服务器中，使用 SHOW STATUS 命令可以查看与当前连接相关的多种状态值。在此基础上，MariaDB 服务器又添加了“Memory_used”状态变量。

```
MariaDB> SHOW STATUS;
+-------------------------------------------------+-------------------+
| Variable_name                                   | Value             |
+-------------------------------------------------+-------------------+
...
| Memory_used                                     | 28872             |
...

MariaDB> SHOW GLOBAL STATUS;
+-------------------------------------------------+-------------------+
| Variable_name                                   | Value             |
+-------------------------------------------------+-------------------+
...
| Memory_used                                     | 178378136         |
...
```

从上述 SHOW STATUS 命令的执行结果可知，该连接当前正在使用 28 872 字节的内存。SHOW GLOBAL STATUS 命令的执行结果中，显示当前正在使用的内存大小为 178 MB，表示 MariaDB 服务器正在使用的全部内存。使用 SHOW STATUS 命令只能查看当前连接中正在使用的内容。若想查看所有线程的内存使用量，只要查看 INFORMATION_SCHEMA 数据库中的 processlist 数据表即可。

```
MariaDB> USE INFORMATION_SCHEMA;
MariaDB> SELECT ID, USER, HOST, MEMORY_USED FROM PROCESSLIST;
+----+------+-----------------+-------------+
| ID | USER | HOST            | MEMORY_USED |
+----+------+-----------------+-------------+
|  4 | root | localhost:55439 |       64536 |
|  3 | root | localhost:54947 |       44648 |
+----+------+-----------------+-------------+
```

若想查看 MariaDB 中各客户端连接使用的内存总量，只需计算 PROCESSLIST 数据表的

MEMORY_USED 列值的总和即可，如下所示。

```
MariaDB> SELECT SUM(MEMORY_USED) FROM PROCESSLIST;
+------------------+
| SUM(MEMORY_USED) |
+------------------+
|           109184 |
+------------------+
```

7.2.5 SHUTDOWN 命令

使用 MySQL 客户端程序或 JDBC 连接 MySQL 服务器时，没有命令可以用于关闭 MySQL 服务器。要关闭 MySQL 服务器，需要使用 mysqladmin 实用程序或类似 SSH 的终端程序登录相应服务器，然后再关闭 MySQL 服务器。MariaDB 10.0.4 开始支持 SHUTDOWN 命令，使用该命令可以关闭远程 MariaDB 服务器。

```
MariaDB> SHUTDOWN;
Query OK, 0 rows affected (0.00 sec)
```

关闭服务器命令使用起来并不难，只要执行 SHUTDOWN 命令即可，如上所示。但执行 SHUTDOWN 命令前，必须首先获得运行 SHUTDOWN 命令的权限。借助 MySQL 服务器默认内置的 mysqladmin 实用程序可以关闭 MySQL 服务器，此时为了执行 mysqladmin shutdown 命令，也需要具有执行 SHUTDOWN 命令的权限。

7.2.6 强制终止用户或查询执行

MySQL 5.6 提供了 KILL 命令，用于立刻终止特定连接或连接中正在执行的查询。MariaDB 对 KILL 命令的功能进行了扩展，使终止命令立即执行，或者等待一定时间后再执行，也提供了强制终止特定用户所有连接的功能。下面学习 MariaDB 服务器提供的几种 KILL 命令的使用方法。

强制终止特定连接

与 MySQL 一样，在 MariaDB 中也可以通过连接的 ID 强制终止特定连接。强制终止连接时，需要将线程 ID 作为参数明确指出，KILL 命令后面的 CONNECTION 关键字可以省略。

```
MariaDB> KILL CONNECTION 11;
MariaDB> KILL 11;
```

强制终止特定连接中的查询

MariaDB 中，也可以仅强制终止特定连接中正在执行的查询而保留该连接。只终止特定连接中正在执行的查询时，可以如下使用 KILL QUERY 命令。与 KILL CONNECTION 一样，使用 KILL QUERY 命令时也要给出线程 ID 作为参数。

```
MariaDB> KILL QUERY 11;
```

使用查询 ID 强制终止特定连接中的查询

MariaDB 中，也可以借助查询 ID 强制终止特定连接中正在执行的查询。与 KILL QUERY 命令一样，也会保留相关连接。

```
MariaDB> KILL QUERY ID 125;
```

然而，MySQL 或 MariaDB 中，使用 SHOW PROCESSLIST 命令无法显示查询 ID。使用查询 ID 终止正在执行的查询前，首先要访问 INFORMATION_SCHEMA 数据库中的 PROCESSLIST 数据表，获取正在执行的查询及其 ID。请注意，上述示例中的“125”不是线程 ID，而是查询 ID。

KILL QUERY 与 KILL QUERY ID 两个命令看上去拥有相同功能，但 KILL QUERY 命令强制终止当前正在执行的查询时，不会查看目标连接正在执行什么查询。也就是说，管理员使用 SHOW PROCESSLIST 查看并执行 KILL QUERY 时，若正在执行不同查询，使用 KILL QUERY 命令会终止该连接中所有正在执行的查询。而使用 KILL QUERY ID 命令仅会终止想要终止的查询。也就是说，使用该命令只会强制终止管理员通过 SHOW PROCESSLIST 或 INFORMATION_SCHEMA 的 PROCESSLIST 命令查看并确认有问题的查询。

强制终止特定用户的连接或查询

MySQL 服务器或 MariaDB 服务器中，有时只需强制终止特定用户创建的连接或特定用户正在执行的查询。特别是 MySQL 服务器或 MariaDB 服务器状态不佳或处于危险中时，经常要这样做。但 MySQL 服务器中无法强制终止特定用户的连接或查询，所以直接开发了用于执行该任务的实用程序。MariaDB 中可以使用基于用户的强制终止功能，如下所示。

```
MariaDB> KILL CONNECTION USER 'matt';
MariaDB> KILL CONNECTION USER 'matt@'%'';

MariaDB> KILL QUERY USER 'matt';
MariaDB> KILL QUERY USER 'matt@'%'';
```

上述示例中，执行第一个命令时不考虑用户账号中的主机部分，而将用户名为 matt 的所有连接强制终止。第二个命令将用户账号中主机部分为 % 且用户名为 matt 的所有连接强制终止。第三个与第四个命令并非用于终止连接，而仅强制终止连接中正在执行的查询。

调整强制终止级别

强制终止特定连接中正在执行的查询时，数据表的数据有时会陷入不一致的状态（Inconsistent state）。特别是 MEMORY 或 MyISAM 以及 Aria 这类不支持事务的存储引擎中，经常会发生这种情况。在 MariaDB 中执行 KILL 命令时，不会强制终止这些影响数据一致性的查询。

```
MariaDB> KILL HARD QUERY 11;
MariaDB> KILL SOFT QUERY 11;
```

第一个 KILL 命令中使用的 HARD 选项用于立刻强制终止 11 号线程中正在执行的查询。有

对数据一致性产生不良影响的 RPAIR 或 ALTER INDEX 等命令时，第二个 KILL 命令中使用 SOFT 选项不会终止 11 号线程中正在执行的查询。

7.2.7 GET DIAGNOSTICS

MySQL 存储过程或函数中，异常处理功能十分不完善。MySQL 5.1 不支持用户直接触发异常的功能，但 MySQL 5.5 开始引入 SIGNAL 与 RESIGNAL 命令，用户可以直接触发异常，并可以捕获（Catch）触发的异常或错误。MySQL 5.5 中，虽然可以如下为特定错误号或所有 SQL 异常定义处理程序（Handler），但无法访问发生错误的错误号或 SQLSTATE 值。

```
DECLARE EXIT HANDLER FOR 1062 SET v_error := "Duplicate entry in table";
DECLARE EXIT HANDLER FOR sqlexception SET v_error := "Generic SQLException";
```

MySQL 5.6 与 MariaDB 10.0.x 开始引入 GET DIAGNOSTICS 命令，借助该命令，存储程序可以访问实际发生的异常的错误号、SQLSTATE 值以及错误信息。下面是使用 GET DIAGNOSTICS 命令的示例，它是一个简单的存储过程，只向 employees 数据表进行插入（INSERT）操作。

```
DELIMITER ;;

CREATE PROCEDURE insertEmployee(
  p_empno INT,
  p_birthdate DATE,
  p_firstname VARCHAR(14),
  p_lastname VARCHAR(16),
  p_gender ENUM('M','F'),
  p_hiredate DATE)
BEGIN
  DECLARE v_sqlstate CHAR(5) DEFAULT '00000';
  DECLARE v_message TEXT;
  DECLARE v_errno INT;

  DECLARE EXIT HANDLER FOR SQLEXCEPTION
  BEGIN
    GET DIAGNOSTICS CONDITION 1
      v_sqlstate = RETURNED_SQLSTATE,
      v_errno=MYSQL_ERRNO,
      v_message = MESSAGE_TEXT;
    IF sqlstate_code <> '00000' THEN
       IF mysql_errno = 1062 THEN     # Duplicate employee no
        SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT='Duplicate employee no';
```

```
        ELSEIF mysql_errno = 1048 THEN
          SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT='Employee no must be not null';
        ELSE
          RESIGNAL;
        END IF;
    END IF;
  END;

  INSERT INTO employees (emp_no, birth_date, first_name, last_name, gender, hire_date)
  VALUES (p_empno, p_birthdate, p_firstname, p_lastname, p_gender, p_hiredate);
END;;
```

该存储过程中，执行INSERT语句时若发生错误，则分析错误内容，然后使用GET DIAGNOSTICS命令将错误信息以一种更友好的方式返回给用户。下面尝试调用该存储过程。

```
MariaDB> CALL insertEmployee(10001, '1997-01-10', 'Matt', 'Lee', 'M', '2014-02-02');
ERROR 1644 (45000): Duplicate employee no

MariaDB> CALL insertEmployee(NULL, '1997-01-10', 'Matt', 'Lee', 'M', '2014-02-02');
ERROR 1048 (45000): Employee no must be not null
```

注意 若不编写存储过程而使用Java、PHP等编程语言编写，为了从MariaDB服务器获得错误号与SQLSTATE，则没有必要非得使用GET DIAGNOSTICS功能。这些编程语言的Exception或Error处理功能中，本身就包含了获取MySQL服务器的错误号或SQLSTATE值的功能。使用GET DIAGNOSTICS命令可以使存储过程以更具弹性的方式进行错误处理。

7.3 开发效率 MariaDB

MariaDB服务器从5.5开始就逐渐添加各种功能，以方便用户开发程序，大幅提高开发效率。其中，较具代表性的是MariaDB 10.0提供的Sequence存储引擎，借助它可以大大增加编写SQL语句的便利性。开发者迫切需要的另一个功能是限制系统过度使用的功能。即使是MariaDB服务器专家，有时也会因为错误执行SELECT查询而造成MariaDB服务器无法正常执行服务查询。此外，MariaDB还一直在逐渐实现许多类似PostgreSQL的RDBMS提供的常用功能。

7.3.1 LIMIT ROWS EXAMINED

MariaDB服务器新增了LIMIT ROWS EXAMINED语法，用户通过其请求的SELECT查询的执行过程中，若比较是否满足条件的记录数超过指定条数，则该查询就会终止执行。请注意，LIMIT ROWS EXAMINED中指定的条数并非要传送给客户的最终结果所含的记录条数，而是

MariaDB 服务器为了产生用户请求的结果而在内部处理的记录条数。请看下列示例查询。

```
MariaDB> SELECT * FROM employees WHERE last_name='Sudbeck' LIMIT ROWS EXAMINED 100;
Empty set, 1 warning (0.00 sec)

MariaDB> SHOW WARNINGS\G
  Level: Warning
   Code: 1931
Message: Query execution was interrupted. The query examined at least 101 rows,
         which exceeds LIMIT ROWS EXAMINED (100).
         The query result may be incomplete.
```

上面示例查询从 employees 数据表查询 last_name 为“Sudbeck”的员工信息，并在查询语句的末尾添加“LIMIT ROWS EXAMINED 100”条件。由于 employees 数据表的 last_name 数据列不存在索引，所以上述示例查询执行全表扫描以查找结果。全表扫描方式下，从 employees 数据表逐条取出记录，检查 last_name 列值是否为 Sudbeck。由于存在“LIMIT ROWS EXAMINED 100”限制条件，试图比较第 101 条记录时，MariaDB 服务器将不会再执行比较操作，而是发出警告并直接终止查询。图 7-5 表示对 employees 数据表进行全表扫描的过程中，比较到第 101 条记录时终止比较操作并返回的处理过程。

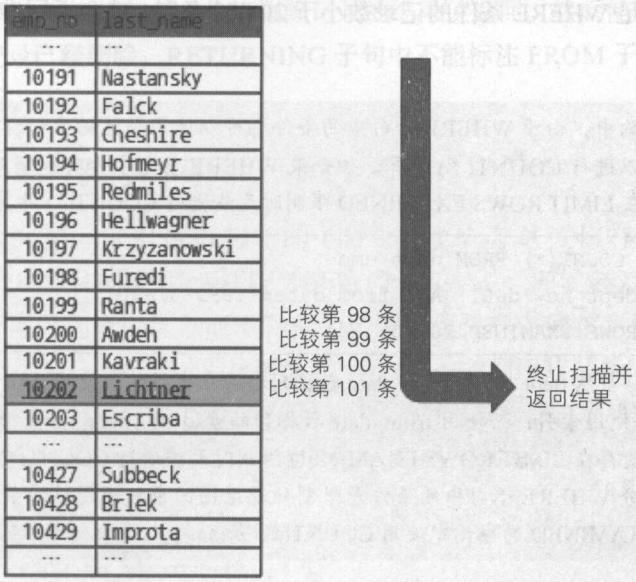


图 7-5 LIMIT ROWS EXAMINED 工作方式

请注意，“LIMIT ROWS EXAMINED n”子句中，查询语句无法使用合适的索引时，返回 n 条记录前，查询可能就会终止。当然，即使 WHERE 子句中的一部分条件使用索引，若 WHERE 子句的所有条件无法使用合适的索引，那么用于比较的记录条数（EXAMINED ROWS）也可能要比查询结果所含的记录数更多。

通过“LIMIT ROWS EXAMINED n”子句可以限制最大记录数，也有可能在所限制的最多记录条数内求得满足条件的所有记录条数。使用 COUNT(*) 的查询中通常是无法限制要计算的记录条数的。当然，可以用子查询的形式创建临时表，查询（SELECT）n 条记录后，在外部再执行 COUNT(*) 操作。但由于记录要复制到临时表，所以这种方法并不高效。下列示例查询使用 LIMIT ROWS EXAMINED 子句。

```
MariaDB> SELECT COUNT(*) FROM dept_emp WHERE dept_no='d001' LIMIT ROWS EXAMINED 20000;
Empty set, 1 warning (0.02 sec)

MariaDB> SELECT COUNT(*) FROM dept_emp WHERE dept_no='d002' LIMIT ROWS EXAMINED 20000;
+----------+
| count(*) |
+----------+
|    17346 |
+----------+
```

上述两个查询用于计算相应部门员工的数量，数量大于 20 000 名时，将不再计算。第一个查询结果为空，由于 LIMIT ROWS EXAMINED 子句中设定的记录条数限制，计数操作终止，故最终返回空集（Empty set）。但从中仍然可知，dept_emp 数据表中符合条件的记录最少在 20 000 条以上[①]。满足 WHERE 条件的记录数小于 20 000 条时，就会返回准确的结果条数，如第二个查询示例所示。

注意 上述两个示例查询中，由于 WHERE 子句中的条件都可以使用合适的索引，所以在记录条数受限制的条件下，仍然可以进行 COUNT(*) 计算。但如果 WHERE 子句中含有如下无法使用索引的条件，那么就如前所述，存在 LIMIT ROWS EXAMINED 限制时无法进行 COUNT(*) 计算。

```
MariaDB> SELECT COUNT(*) FROM dept_emp
         WHERE dept_no='d001' AND from_date='1999-01-10'
         LIMIT ROWS EXAMINED 20000;
```

上述示例查询中，虽然 dept_no 与 from_date 数据列都存在索引，但使用 dept_no 数据列的索引时，from_date 条件就无法使用索引；而使用 from_date 数据列的索引时，dept_no 数据列的条件就不能使用索引。这种过滤条件使存在 LIMIT ROWS EXAMINED 限制时无法进行 COUNT(*) 计算。

如果无法明确区分 WHERE 子句中的条件是限制处理范围的条件还是过滤条件，那么最好不要在带有 LIMIT ROWS EXAMINED 的查询中使用 COUNT(*) 功能。

7.3.2 DELETE...RETURNING...

MySQL 服务器中，执行 DELETE 语句时，无法查看删除的记录。但 PostgreSQL 提供了选择性获取所删记录数据列的功能，MariaDB 10.0.5 添加了该功能。若想获取删除记录的数据列，

① 当然，在此情形下查看警告信息可以获知警告代码（1931）以及更详细的警告信息，并从中获得更加准确的结果。

只要在 DELETE 语句的末尾添加 RETURNING 子句即可。RETURNING 子句中不能使用集合函数与子查询等。请看下列简单示例。

```
MariaDB> INSERT INTO EMPLOYEES VALUES (1, '1985-01-21','Matt','Lee','M','2014-01-10');

MariaDB> DELETE FROM employees WHERE first_name='Matt' AND last_name='Lee'
         RETURNING emp_no, first_name, last_name;

+--------+------------+-----------+
| emp_no | first_name | last_name |
+--------+------------+-----------+
|      1 | Matt       | Lee       |
+--------+------------+-----------+
1 row in set (0.05 sec)

MariaDB> SELECT * FROM employees WHERE first_name='Matt' AND last_name='Lee';
Empty set (0.00 sec)
```

为了进行删除测试，上述示例首先向 employees 数据表插入 1 条记录，然后执行带有 RETURNING 子句的 DELETE 语句。RETURNING 子句罗列了 emp_no、first_name、last_name 三个数据列，执行 DELETE 语句后，作为删除记录的结果值返回。然后使用 SELECT 语句查询该记录，可以发现记录已经删除。RETURNING 子句中不能标注 FROM 子句，只能是从删除的记录中获取的数据列。

7.3.3 以微秒为单位保存时间

从 MariaDB 5.3 开始，可以以微秒（1/1 000 000 秒）为单位保存时间信息。原有的 DATETIME 或 TIMESTAMP 类型中无法定义精度（Precision），即使定义也没有太大意义。从 MariaDB 5.3 开始，可以选择以毫秒或微秒作为精度单位保存时间信息。下列示例创建 3 个 DATETIME 类型的数据列——精度分别设置为 0、3、6——并利用其保存时间信息。

```
MariaDB> CREATE TABLE tb_microsecond(
           fd1 DATETIME(0), fd2 DATETIME(3), fd3 DATETIME(6));

MariaDB> INSERT INTO tb_microsecond VALUES (NOW(6), NOW(6), NOW(6));
MariaDB> SELECT * FROM tb_microsecond;

+---------------------+-------------------------+----------------------------+
| fd1                 | fd2                     | fd3                        |
+---------------------+-------------------------+----------------------------+
| 2014-02-02 14:49:56 | 2014-02-02 14:49:56.298 | 2014-02-02 14:49:56.298633 |
+---------------------+-------------------------+----------------------------+
```

数据列 fd1 的精度为 0，它与不带精度创建的 DATETIME 数据列是一样的。而将精度分别设置为 3 或 6，即意味着可以以毫秒或微秒为单位保存或访问时间信息。此处需要注意，用于获取当前时间的 NOW() 函数也要将精度设置为 6。获取当前时间时将精度设为 6，即调用 NOW(6) 函数，即可获取以微秒为精度单位的当前时间。下面使用不带精度的 NOW() 函数，将当前时间插入 tb_microsecond 数据表，如下所示。

```
MariaDB> INSERT INTO tb_microsecond VALUES (NOW( ), NOW( ), NOW( ));
MariaDB> SELECT * FROM tb_microsecond;
+---------------------+-------------------------+-------------------------+
| fd1                 | fd2                     | fd3                     |
+---------------------+-------------------------+-------------------------+
| 2014-02-02 14:49:41 | 2014-02-02 14:49:41.000 | 2014-02-02 14:49:41.000000 |
+---------------------+-------------------------+-------------------------+
```

如上述示例所示，调用不带精度的 NOW() 函数只能获得以秒为精度单位的当前时间。即使 DATETIME 类型数据列（如 fd2 与 fd3）的精度很高，其内部保存的时间在秒之后的部分也只能填充为 0。由于可以以毫秒或微秒为单位保存时间信息，所以大部分与日期、时间有关的函数都做出了改进，它们一般都可以接收精度参数。

```
MariaDB> SELECT NOW(0), NOW(3), NOW(6);
+---------------------+-------------------------+-------------------------+
| NOW(0)              | NOW(3)                  | NOW(6)                  |
+---------------------+-------------------------+-------------------------+
| 2014-02-02 15:01:11 | 2014-02-02 15:01:11.948 | 2014-02-02 15:01:11.948278 |
+---------------------+-------------------------+-------------------------+

MariaDB> SELECT UNIX_TIMESTAMP(NOW(6));
+--------------------------+
| UNIX_TIMESTAMP(NOW(6)) |
+--------------------------+
|      1391320977.307304 |
+--------------------------+

MariaDB> SELECT NOW(6), DATE_SUB(NOW(6), INTERVAL 100000 MICROSECOND);
+----------------------------+-----------------------------------------+
| NOW(6)                     | DATE_SUB(NOW(6), INTERVAL 100000 MICROSECOND) |
+----------------------------+-----------------------------------------+
| 2014-02-02 15:04:04.762162 | 2014-02-02 15:04:04.662162              |
+----------------------------+-----------------------------------------+
```

第二个示例调用了 UNIX_TIMESTAMP 函数，该函数不带参数时，返回当前时间点的时间戳，但不能返回以毫秒或微秒为单位的时间戳。若想使用 UNIX_TIMESTAMP 函数获取带有毫

秒或微秒的当前时间，就要将 NOW(3) 或 NOW(6) 作为参数传入函数。DATE_SUB 或 DATE_ADD 函数对时间进行加减时，也可以将时间单位指定为微秒，如上述第三个示例所示。

注意 复制备份系统中，若 MariaDB 为主服务器，MySQL 为从服务器，则最好不要使用 MariaDB 服务器中以微秒为单位的 DATETIME 类型。MySQL 5.5 尚不支持以微秒为单位的数据类型或函数，MySQL 5.6 采用的方式与 MariaDB 不同，将以微秒为精度单位的时间记录到二进制日志。若在 MariaDB 与 MySQL 服务器构成的复制备份系统中使用了 MariaDB 服务器中以微秒为单位的 DATETIME 类型，那么即使未发生复制错误，以微秒为单位的时间信息也不会复制到用作从服务器的 MySQL。

7.3.4 设置 DATETIME 类型的默认值

MySQL 5.5 或 MariaDB 5.5 之前，虽然可以将当前时间设置为 TIMESTAMP 类型的默认值，但不能设置为 DATETIME 类型数据列的默认值。从 MariaDB 10.0 开始，用户也可以将当前时间设置为 DATETIME 类型数据列的默认值。请看下列示例。

```
MariaDB> CREATE TABLE tb_datetime (
           fd1 INT PRIMARY KEY,
           fd2 DATETIME DEFAULT CURRENT_TIMESTAMP);

MariaDB> INSERT INTO tb_datetime (fd1) VALUES (1);
MariaDB> SELECT * FROM tb_datetime;
+-----+---------------------+
| fd1 | fd2                 |
+-----+---------------------+
|   1 | 2014-02-02 14:40:50 |
+-----+---------------------+
```

由于 MySQL 中 FRM 文件自身特性，只能存在 1 个将 CURRENT_TIMESTAMP 设置为默认值的 TIMESTAMP 类型的数据列。但 DATETIME 类型中不存在这样的限制，所以可以创建多个以当前时间为默认值的 DATETIME 类型的数据列。此外，从 MariaDB 5.3 开始，DATETIME 与 TIMESTAMP 类型都可以保存以微秒为单位的时间。这样，用户保存日期与时间信息时的可选范围大大增加。

7.3.5 扩展正则表达式功能

MySQL 5.5 与 5.6 服务器使用兼容 POSIX 的正则表达式库（http://www.arglist.com/regex/），而 MariaDB 10.0.5 却做出改变，使用 PCRE（Perl Compatible Regular Expressions）正则表达式库。MariaDB 服务器中，使用 PCRE 正则表达式可以大大提升 REGEXP() 函数与 RLIKE() 函数的性能，同时可以使正则表达式的使用更加灵活多样。由于 PCRE 正则表达式兼容 POSIX 正则表达式中的语法，所以可以直接使用原来 MySQL 中开发的使用正则表达式的应用程序的 SQL

语句。若想简单了解 MariaDB 的 PCRE 正则表达式语法，请参考 MariaDB 正则表达式介绍页面，会有很大帮助。

- https://mariadb.com/kb/en/regular-expressions-overview/
- https://mariadb.com/kb/en/pcre-regular-expressions/

此外，在 MySQL 中使用正则表达式只能针对全部匹配或部分匹配进行确认。从 MariaDB 10.0.5 开始，在原有匹配功能基础上，添加并完善了利用正则表达式进行位置定位与字符串替换的功能。MariaDB 服务器添加的正则表达式函数有如下三种。

REGEXP_REPLACE（字符串，正则表达式，替换字符串）

该函数从第一个参数给出的字符串中搜索与正则表达式相匹配的部分，将其替换为第三个参数给出的替换字符串，然后返回结果字符串。此时，所有与正则表达式匹配的部分都会被替换并返回。若想忽略数据列中默认设置的大小写区分规则，而应用新的大小写比较规则，只要在正则表达式中使用“(?!)”与“(?-!)”选项即可。“(?-!)”选项用于强制区分大小写，“(?!)”选项用于在不区分大小写的情形下进行比较。

REGEXP_INSTR（字符串，正则表达式）

REGEXP_INSTR 函数从给定字符串中查找与正则表达式相匹配的字符串的位置，并将其返回。返回字符串的位置时，不是以字节为单位，而是以字符为单位。多字节文字（以汉语为代表的亚洲文字）中，使用正则表达式以字节为单位查找字符串位置时，要先将类型转换为 BINARY。

REGEXP_SUBSTR(字符串，正则表达式)

该函数仅返回第一个参数给定的字符串的子串，该子串与正则表达式相匹配。与 REGEXP_REPLACE 函数类似，区分大小写是原数据列的规则，但可以使用“(?!)”与“(?-!)”选项，对大小写区分规则进行重定义。

```
--// 使用正则表达式进行 REPLACE
MariaDB> SELECT REGEXP_REPLACE('ab12cd','[0-9]','') AS remove_digits;
-> abcd

MariaDB> SELECT
         REGEXP_REPLACE('<html><head><title>title</title><body>body</
body></htm>', '<.+?>',' ')
           AS tag_removed;
-> title  body

--// 在正则表达式 REPLACE 中区分大小写
MariaDB> SELECT REGEXP_REPLACE('ABC','(?-i)b','-') AS force_case_sensitive;
-> ABC

--// 在正则表达式 REPLACE 中不区分大小写
MariaDB> SELECT REGEXP_REPLACE(BINARY 'ABC','(?i)b','-') AS force_case_insensitive;
```

```
-> A-C

--// 使用正则表达式查找字符串位置
MariaDB> SELECT REGEXP_INSTR('BJÖRN','N') AS pos;
-> 5

--// 使用正则表达式查找字符串位置（不是字符位置而是字节位置）
MariaDB> SELECT REGEXP_INSTR(BINARY 'BJÖRN','N') AS binary_pos;
-> 6

--// 使用正则表达式获取字符串子串
MariaDB> SELECT REGEXP_SUBSTR('ab12cd','[0-9]+');
-> 12
```

提示 MySQL 5.6 中使用的 POSIX 正则表达式与 MariaDB 10.0 中使用的 PCRE 正则表达式在基本语法上类似，但对于复杂形式的表达式，二者在语法或功能上有差异。限于篇幅，本书无法罗列两种正则表达式的全部差别，感兴趣的读者可以参考下列网页，详细学习两种正则表达式在功能与语法上的不同。

http://www.regular-expressions.info/refbasic.html

打开浏览器，进入上述页面，左侧显示的是正则表达式的功能列表，如图 7-6 所示。

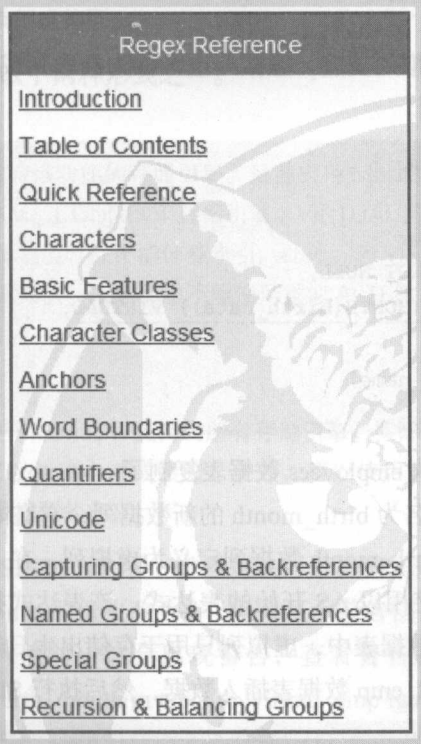


图 7-6 正则表达式的功能列表

图 7-6 所示的功能列表中，点击“Basic Features”或“Character Classes”等项目，进入两种正则表达式的功能比较页面，如图 7-7 所示。页面标题下有两个选择框，可以从中选择要比较的正则表达式。在图 7-7 左侧选择框选择“PCRE”、右侧选择框选择“POSIX ERE”，进行比较。

Regular Expression Reference: Basic Features

PCRE ▾ POSIX ERE ▾

Feature	Syntax	Description	Example	PCRE	POSIX ERE
Dot	. (dot)	Matches any single character except line break characters. Most regex flavors have an option to make the dot match line break characters too.	. matches x or (almost) any other character	YES	YES
Alternation	| (pipe)	Causes the regex engine to match either the part on the left side, or the part on the right side. Can be strung together into a series of options.	abc|def|xyz matches abc, def or xyz	YES	YES
Alternation	| (backslash pipe)	Causes the regex engine to match either the part on the left side, or the part on the right side. Can be strung together into a series of options.	abc|def|xyz matches abc, def or xyz	no	no

图 7-7 正则表达式的功能比较

7.3.6 虚拟列

MariaDB 5.2 开始提供虚拟列功能。虚拟列也常称为抽取列或派生列，其列值由其他数据列自动设置。下面介绍简单示例。

```
CREATE TABLE tb_virtual_emp (
  emp_no int(11) NOT NULL,
  birth_date date NOT NULL,
  first_name varchar(14) NOT NULL,
  birth_month TINYINT AS (MONTH(birth_date)) VIRTUAL,
  PRIMARY KEY (emp_no),
  KEY ix_firstname (first_name)
) ENGINE=InnoDB;
```

tb_virtual_emp 数据表只从 employees 数据表复制了 emp_no、birth_date、first_name 数据列。tb_virtual_emp 数据表还包含名为 birth_month 的新数据列，它在 employees 数据表中并不存在。借助 VIRTUAL 关键字，将 birth_month 数据列定义为虚拟列。在 MariaDB 服务器中定义虚拟列时，列定义的后半部分总要使用以 AS 开始的表达式。该表达式提供计算公式，以对虚拟列值进行初始化。tb_virtual_emp 数据表中，虚拟列只用于存储出生日期（birth_date）中的月份信息（MONTH 函数）。向 tb_virtual_emp 数据表插入数据，然后执行 SELECT 查询。

```
MariaDB> INSERT INTO tb_virtual_emp(emp_no, birth_date, first_name)
         SELECT emp_no, birth_date, first_name FROM employees;
```

```
MariaDB> SELECT * FROM tb_virtual_emp LIMIT 5;
+--------+------------+------------+-------------+
| emp_no | birth_date | first_name | birth_month |
+--------+------------+------------+-------------+
|  10001 | 1953-09-02 | Georgi     |           9 |
|  10002 | 1964-06-02 | Bezalel    |           6 |
|  10003 | 1959-12-03 | Parto      |          12 |
|  10004 | 1954-05-01 | Chirstian  |           5 |
|  10005 | 1955-01-21 | Kyoichi    |           1 |
+--------+------------+------------+-------------+
```

虽然 INSERT 语句中并没有对 birth_month 数据列进行初始化，也没有为其设置值，但从查询结果可以看到，birth_month 数据列值已经初始化。

下面介绍虚拟列的几个特征。首先，MariaDB 的虚拟列有如下几个限制条件。

- 虚拟列表达式要定义在 252 个字符以内。
- 与子查询一样，不能使用引用外部数据表数据的表达式。
- 不能使用含有用户自定义函数（UDF）与存储函数的表达式。
- 常量表达式不能用作虚拟列的表达式。
- 虚拟列表达式中不能使用其他虚拟列。

定义虚拟列时，可以使用以下两种形式之一。

VIRTUAL

VIRTUAL 虚拟列只能用于支持该功能的存储引擎。目前支持 MariaDB 的 VIRTUAL 虚拟列的存储引擎有 InnoDB、Aria、MyISAM、CONNECT 存储引擎。VIRTUAL 虚拟列并不会将值保存到实际的存储器，每次访问它时才会计算其值，故不能创建索引。此外，若 VIRTUAL 虚拟列不用于 SELECT 查询，则实际在内部也不会计算其值。此外，也不能使用类似 ALTER TABLE MODIFY 或 CHANGE 的更改数据列定义的 DDL 语句。

PERSISTENT

PERSISTENT 虚拟列可以用于 MariaDB 支持的所有存储引擎。实际的存储引擎中，PERSISTENT 虚拟列被管理为数据列，故可以创建主键或索引。此外，可以使用类似 ALTER TABLE MODIFY 或 CHANGE 的 DDL 语句更改数据列的定义。

最后，VIRTUAL 或 PERSISTENT 两个虚拟列执行 INSERT 操作后都不能再修改其值。下列示例尝试将虚拟列 birth_month 的值更改为 1，但只显示警告信息，并未实际改变记录的虚拟列值。像这样试图更改虚拟列值时，就会出现警告，查看警告信息内容为“1906, The value specified for computed column birth_month in table tb_virtual_emp ignored”。

```
MariaDB> SELECT * FROM tb_virtual_emp WHERE emp_no=10001;
```

```
+--------+------------+------------+-------------+
| emp_no | birth_date | first_name | birth_month |
+--------+------------+------------+-------------+
|  10001 | 1953-09-02 | Georgi     |           9 |
+--------+------------+------------+-------------+

MariaDB> UPDATE tb_virtual_emp SET birth_month=1 WHERE emp_no=10001;
Query OK, 0 rows affected, 1 warning (0.02 sec)
Rows matched: 1  Changed: 0  Warnings: 1

MariaDB> SELECT * FROM tb_virtual_emp WHERE emp_no=10001;
+--------+------------+------------+-------------+
| emp_no | birth_date | first_name | birth_month |
+--------+------------+------------+-------------+
|  10001 | 1953-09-02 | Georgi     |           9 |
+--------+------------+------------+-------------+
```

7.3.7 动态列

MariaDB 5.3 开始提供动态列功能，用于实现对 NoSQL 形态的数据存储与访问。动态列定义一个大容量的数据列，在其中可以定义并使用多个临时数据列。也就是说，MariaDB 服务器虽然只能拥有一个物理数据列，但物理数据列内部可以同时使用多个逻辑数据列。下面通过简单示例学习动态列的使用方法。

```
MariaDB> CREATE TABLE tb_dynamic_emp (
  emp_no INT NOT NULL,
  dyna_cols BLOB,
  PRIMARY KEY(emp_no)
);

MariaDB>INSERT INTO tb_dynamic_emp (emp_no, dyna_cols)
        VALUES (10001,
          COLUMN_CREATE('birth_date','1953-09-02',
                        'first_name','Georgi',
                        'last_name','Facello',
                        'gender','M',
                        'hire_date','1986-06-26')
);

MariaDB> SELECT * FROM tb_dynamic_emp\G
*************************** 1. row ***************************
   emp_no: 10001
```

```
dyna_cols:  ,     # ?S" genderhire_datelast_namebirth_datefirst_
name!M!1986-06-26!Facello!1953-09-02!Georgi
```

上述示例中，除 employees 数据表的主键 emp_no 外，其余数据列值都像 NoSQL 一样以键值对（Key/Value）的形式存储于 dyna_cols。访问 tb_dynamic_emp 数据表时若不使用任何实用工具函数，dyna_cols 列值就会发生混乱。也就是说，使用 COLUMN_CREATE() 函数会先生成二进制数据，然后保存，所以为要用作动态列的物理列指定类型时，必须将其设为 VARBINARY 或 BLOB 类型，这两个类型都是二进制类型。若用作动态列的物理列类型被定义为 VARCHAR 或 TEXT，那么执行 INSERT 或 UPDATE 操作时就会发生如下错误。

```
Error Code : 1366
Message : Incorrect string value: '\xD3\x00\x18\x00S\x01...' for column
'dyna_cols' at row 1
```

保存、更改或删除动态列时，必须使用实用工具函数。MariaDB 提供了以下实用工具函数处理动态列值。

COLUMN_CREATE

COLUMN_CREATE 函数用于初始化动态列并存储列值，它拥有多个参数，这些参数由成对的“列名”与“列值”组成。也就是说，COLUMN_CREATE 函数所需参数总由“列名”与“列值”对组成，故需要偶数个参数，如下所示。此外，列值不仅可以保存为字符串，也可以保存为 AS TYPE 指定的类型，比如 AS DATE。

```
COLUMN_CREATE('column_name1', 'column_value1',  'column_name2', 'column_value2'
AS TYPE, ..)
```

COLUMN_ADD

该函数用于向已经拥有值的物理动态列存储新的逻辑动态列。与 COLUMN_CREATE 函数类似，COLUMN_ADD 函数也需要参数，但必须先明确指出物理动态列名。

```
COLUMN_ADD(dyna_cols,  'column_name3', 'column_value3', ...);
```

COLUMN_GET

该实用工具函数从物理动态列获取逻辑列值，它需要两个参数，并且总是返回 1 个逻辑列值。

```
COLUMN_GET(dyna_cols, 'column_name1' AS DATE)
```

使用 COLUMN_GET 函数获取逻辑列值时，必须明确指出类型，可以使用的类型如下所示。

- BINARY[(N)]
- CHAR[(N)]
- DATE
- DATETIME[(D)]
- DECIMAL[(M[,D])]

- DOUBLE[(M,D)]
- INTEGER
- SIGNED [INTEGER]
- TIME[(D)]
- UNSICNED [INTEGER]

COLUMN_DELETE

COLUMN_DELETE 函数用于删除逻辑动态列，每次可以删除多个逻辑列。

```
COLUMN_DELETE(dyna_cols, 'column_name1', 'column_name2', ...)
```

COLUMN_EXISTS

该函数检查指定名称的逻辑动态列是否存在，每次只能检查 1 个逻辑列存在与否。

```
COLUMN_EXISTS(dyna_cols, 'column_name1')
```

COLUMN_LIST

该实用工具函数仅用于返回物理动态列拥有的所有逻辑动态列的名称。

```
COLUMN_LIST(dyna_cols)
```

COLUMN_CHECK

该实用工具函数检测物理动态列拥有的值是否被正常打包。若其返回值为 1，则表示逻辑列被正常打包；若返回 0，则表示出现错误。

```
COLUMN_CHECK(dyna_cols)
```

COLUMN_JSON

该函数以 JSON 格式返回物理动态列拥有的所有逻辑动态列的名称与值。MariaDB 的动态列可以嵌套使用，对于嵌套动态列，COLUMN_JSON 函数最多只能进行 10 级解码。

```
COLUMN_JSON(dyna_cols)
```

以上就是用于访问动态列的实用工具函数，下面通过几个简单示例学习各函数的使用方法。

```
MariaDB> UPDATE tb_dynamic_emp SET dyna_cols=COLUMN_ADD(dyna_cols, 'country','Korea');
Query OK, 1 row affected (0.05 sec)
Rows matched: 1  Changed: 1  Warnings: 0

MariaDB> SELECT emp_no, COLUMN_GET(dyna_cols, 'country' as CHAR) as v_country
         FROM tb_dynamic_emp;
+--------+-----------+
| emp_no | v_country |
```

```
+--------+-----------+
|  10001 | Korea     |
+--------+-----------+

MariaDB> UPDATE tb_dynamic_emp SET dyna_cols=COLUMN_DELETE(dyna_cols, 'country');
Query OK, 1 row affected (0.08 sec)
Rows matched: 1  Changed: 1  Warnings: 0

MariaDB> SELECT emp_no, COLUMN_EXISTS(dyna_cols, 'country') as existence
FROM tb_dynamic_emp;
+--------+-----------+
| emp_no | existence |
+--------+-----------+
|  10001 |         0 |
+--------+-----------+

MariaDB> SELECT emp_no, COLUMN_LIST(dyna_cols) as col_list FROM tb_dynamic_emp;
+--------+-------------------------------------------------------------------+
| emp_no | col_list                                                          |
+--------+-------------------------------------------------------------------+
|  10001 | 'gender','hire_date','last_name','birth_date','first_name'        |
+--------+-------------------------------------------------------------------+

MariaDB> SELECT emp_no, COLUMN_CHECK(dyna_cols) as is_valid FROM tb_dynamic_emp;
+--------+----------+
| emp_no | is_valid |
+--------+----------+
|  10001 |        1 |
+--------+----------+

MariaDB> SELECT emp_no, COLUMN_JSON(dyna_cols) as is_valid FROM tb_dynamic_emp\G
*************************** 1. row ***************************
  emp_no: 10001
is_valid: {"gender":"M","hire_date":"1986-06-26","last_name":"Facello"
,"birth_date":"1953-09-02","first_name":"Georgi"}
```

MariaDB 的动态列有两个限制条件。

(1) 一个物理动态列最多可以保存 65 535 个逻辑列。

(2) 物理动态列存储值的字节数由 max_allowed_packet 系统变量设定。

7.4 分区 MariaDB MySQL

与之前版本相比，MySQL 5.6 与 MariaDB 10.0 的分区（Partition）功能其实没有太大变化，几乎没有对分区表的性能或分区表拥有的诸多限制条件进行改善或更改。尽管如此，MySQL 5.6 与 MariaDB 10.0 中使用分区表时还是有一些改善，比如分区最大数量由 1024 个增加为 8192 个。此外还添加了显式指定分区以及导入导出分区表的功能，下面进行详细讲解。

7.4.1 显式指定分区

MySQL 5.6 与 MariaDB 10.0 中添加了显式指定目标分区的功能，用户可以直接明确指定要在 SQL 语句中使用的目标分区。下面列出的 SQL 语句添加了新语法，通过这些语法，用户可以明确指定要使用的分区。

- SELECT
- DELETE
- INSERT
- REPLACE
- UPDATE
- LOAD DATA
- LOAD XML

MySQL 5.5 或 MariaDB 5.5 中，分区修剪（Partition Pruning）用于帮助优化器自动选取所需分区，但该功能只能用于 SELECT 查询，而显式分区指定功能在大部分 DML 语句与 SELECT 查询语句中都可以使用。若想使用显示分区指定功能，必须在 FROM 子句的数据表后使用 PARTITION(partition_name1, partition_name2, ...) 子句。此时，PARTITION 子句必须在其所属数据表名后明确标注。若 PARTITION 子句中不存在指定分区，则 MariaDB 服务器就会产生“partition 'partition_ name1' doesn't exist”错误。

1. 显式分区指定的使用方法

下面通过简单示例学习显式分区指定的使用方法。

```
MariaDB> CREATE TABLE tb_partition_employees(
  emp_no int NOT NULL,
  birth_date date NOT NULL,
  first_name varchar(14) NOT NULL,
  last_name varchar(16) NOT NULL,
  gender enum('M','F') NOT NULL,
  hire_date date NOT NULL
) ENGINE=InnoDB
PARTITION BY RANGE COLUMNS(hire_date)(
  PARTITION p0 VALUES LESS THAN ('1990-01-01'),
```

```
  PARTITION p1 VALUES LESS THAN ('2000-01-01'),
  PARTITION p2 VALUES LESS THAN ('2010-01-01')
);

MariaDB> INSERT INTO tb_partition_employees SELECT * FROM employees;
Query OK, 300024 rows affected (6.73 sec)
Records: 300024  Duplicates: 0  Warnings: 0

MariaDB> SELECT COUNT(*) FROM tb_partition_employees;
+----------+
| COUNT(*) |
+----------+
|   300024 |
+----------+

MariaDB> SELECT COUNT(*) FROM tb_partition_employees PARTITION(p0);
+----------+
| COUNT(*) |
+----------+
|   164797 |
+----------+

MariaDB> SELECT COUNT(*) FROM tb_partition_employees PARTITION(p1, p2);
+----------+
| COUNT(*) |
+----------+
|   135227 |
+----------+
```

上述示例中，tb_partition_employees 数据表依据员工入职日期应用了范围分区（Range Partition），每个分区拥有 10 年间入职员工的信息。之后从 employees 数据表复制了其中所有数据记录，共 300 024 条。查询分区数据时并未使用 WHERE 条件语句，而是直接通过“PARTITION(p0)”与“PARTITION(p1)”将要查询的分区明确标注，这样即可访问指定分区中的记录。

除 SELECT、DELETE、UPDATE 之外的其他所有 SQL 语句中，使用显式分区指定功能时，若要存储的记录不符合指定分区的分区标准，就会产生如下错误。当然，由于 SELECT、DELETE、UPDATE 只在指定分区中查找数据，故只返回“无匹配记录”信息。

```
MariaDB> INSERT INTO tb_partition_employees PARTITION(p0)
         VALUES (1, '1984-01-12', 'Matt', 'Lee', 'M', '2009-01-01');
ERROR 1748 (HY000): Found a row not matching the given partition set
```

```
MariaDB> SELECT * FROM tb_partition_employees WHERE emp_no=10001;
+--------+------------+------------+-----------+--------+------------+
| emp_no | birth_date | first_name | last_name | gender | hire_date  |
+--------+------------+------------+-----------+--------+------------+
|  10001 | 1953-09-02 | Georgi     | Facello   | M      | 1986-06-26 |
+--------+------------+------------+-----------+--------+------------+

MariaDB> SELECT * FROM tb_partition_employees PARTITION(p2) WHERE emp_no=10001;
Empty set (0.00 sec)

MariaDB> UPDATE tb_partition_employees PARTITION(p2) SET gender='F' WHERE emp_no=10001;
Query OK, 0 rows affected (0.01 sec)
Rows matched: 0  Changed: 0  Warnings: 0

MariaDB> UPDATE tb_partition_employees PARTITION(p0) SET gender='F' WHERE emp_no=10001;
Query OK, 1 row affected (0.16 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

上述第一个 INSERT 语句将插入目标明确指定为 p0 分区（该分区只保存 hire_date 小于“1990-01-01”的记录），并试图将 hire_date 列值为“2009-01-01”的记录插入其中。也就是说，指定的分区与要保存的数据不匹配，此时会产生“Found a row not matching the given partition set”错误。第二个 SELECT 查询中，访问 emp_no=10001 的员工时并未指定分区，可以正常访问想要的记录。第三个 SELECT 查询中，在 p2 分区搜索 emp_no=10001 的员工，但工号为 10001 的员工实际存储在 p0 分区中，故返回的结果为“Empty set”。第四个与第五个示例查询的 UPDATE 语句中，像 SELECT 一样将分区指定错误时，就无法查找相应记录，最终导致无法成功更新目标记录。

从上述示例可以看到，显式指定分区的 SQL 语句中，首先要进行记录查找的 SQL（SELECT、UPDATE、DELETE）虽然都不会引发错误，但也无法执行想要的操作。并且，需要插入记录的查询中，若均未指定合适的分区，SQL 语句本身就会执行失败。REPLACE 语句最后也要执行插入操作，所以会引发错误。

2. 显式分区指定功能的用途

优化器无法正常处理分区修剪时，使用显式分区指定功能可以将要使用的分区明确标注。但迄今为止，我还没有遇到优化器无法正常对分区修剪进行优化处理的情形。显式分区指定功能还可用于 WHERE 条件语句含有函数的情形。下面通过实际的 SQL 语句进行分析学习。

```
--// 创建要在 WHERE 条件句中使用的函数
CREATE FUNCTION getDate ( )
  RETURNS DATE
```

```
  NOT DETERMINISTIC
BEGIN
  RETURN '2000-01-02';
END;;

MariaDB> EXPLAIN PARTITIONS
          SELECT * FROM tb_partition_employees WHERE hire_date=getDate( )\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: tb_partition_employees
   partitions: p0,p1,p2
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 3
        Extra: Using where

MariaDB> EXPLAIN PARTITIONS
           SELECT * FROM tb_partition_employees PARTITION(p2) WHERE
hire_date=getDate( )\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: tb_partition_employees
   partitions: p2
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 2
        Extra: Using where
```

上述示例的两个 SELECT 查询中，WHERE 条件子句都使用了 getDate() 函数，请注意，此处的 getDate() 函数定义为 NOT DETERMINISTIC。若 getDate() 函数定义为 DETERMINISTIC，则未明确指定分区的第一个 SELECT 查询中也会适当进行分区修剪，使之仅使用 p2 分区。但此处的 getDate() 函数并非 DETERMINISTIC，所以优化器无法预测 getDate() 函数的结果。这样，第一个 SELECT 查询只能搜索所有分区。此时，第二个查询由于明确指定只使用 PARTITION(p2) 分区，所以优化器对查询进行优化，使搜索范围只限制在 p2 分区。

注意 上述示例中，getDate() 函数定义为 NOT DETERMINISTIC，并且用于 SELECT 查询的 WHERE 条件句。其实，我不建议各位将定义为 NOT DETERMINISTIC 的函数用作 WHERE 语句中的条件。若对存储函数或存储过程的 DETERMINISTIC 与 NOT DETERMINISTIC 选项不熟悉，则请尽量将存储程序定义为 DETERMINISTIC。

7.4.2 交换分区表空间

MySQL 5.6 与 MariaDB 10.0 添加了交换（Exchange）分区表空间的功能。此处所说的“交换”是指切换（Swapping）两个数据表空间。交换分区表空间时，将分区表空间与非分区的单一表空间相互切换。要交换表空间需要具备以下几个条件。

- 分区表与要切换表空间的数据表必须具有相同结构。
- 拥有要切换表空间的数据表不能为临时表。
- 要切换的数据表必须满足相应分区的参考列条件（分区范围）。
- 进行切换的两个数据表与其他数据表必须无参照关系（Foreign Key）。
- 对相关数据表必须拥有 ALTER、INSERT、CREATE、DROP 权限。

进行表空间切换时，若未调用原有表的触发器，且存在 AutoIncrement 数据列，则切换完成后，AutoIncrement 值会再次初始化。此处所说的“初始化”并非指再次从 1 开始，而是进行重设，使之从数据表的最大值开始。比如，AutoIncrement 值为 100，数据表的 AutoIncrement 数据列拥有的最大值为 90，切换完成后，下一个 AutoIncrement 并不是 101，而是 91。

该功能看上去没有任何用处，但数据仓库系统中需要移动大量数据的条件有很多。不仅如此，提供在线事务服务的 OLTP 服务器中，该功能经常用于检查或管理作业。此外，该功能与“复制表空间”（6.3.1）结合，可以将分区表空间复制到其他 MySQL 服务器或 MariaDB 服务器。前面所讲的“复制表空间”功能只能用于未分区的数据表，所以没有可以复制分区表空间的功能。但若使用分区表空间切换功能，将分区表空间变换为单一数据表，之后即可复制单一数据表的表空间。

下面介绍分区表空间切换示例。首先准备要切换表空间的分区表和任意数据表。

```
--// 创建分区表
MariaDB> CREATE TABLE tb_partition_employees(
  emp_no int NOT NULL,
  birth_date date NOT NULL,
  first_name varchar(14) NOT NULL,
  last_name varchar(16) NOT NULL,
  gender enum('M','F') NOT NULL,
  hire_date date NOT NULL
) ENGINE=InnoDB
PARTITION BY RANGE COLUMNS(hire_date)(
  PARTITION p0 VALUES LESS THAN ('1990-01-01'),
```

```
  PARTITION p1 VALUES LESS THAN ('2000-01-01'),
  PARTITION p2 VALUES LESS THAN ('2010-01-01')
);

--// 将 employees 数据表的数据复制到 tb_partition_employees 数据表
MariaDB> INSERT INTO tb_partition_employees SELECT * FROM employees;

--// 创建切换中要使用的示例表
MariaDB> CREATE TABLE tb_swap(
  emp_no int NOT NULL,
  birth_date date NOT NULL,
  first_name varchar(14) NOT NULL,
  last_name varchar(16) NOT NULL,
  gender enum('M','F') NOT NULL,
  hire_date date NOT NULL
) ENGINE=InnoDB;

MariaDB> INSERT INTO tb_swap VALUES (1, '1985-01-12', 'Matt', 'Lee', 'M', '1989-10-10');
```

向 tb_swap 数据表插入的 1 条示例记录的 hire_date 为“1989-10-10”，所以该记录位于 tb_partition_employees 分区表的 p0 分区。切换表空间前，要先确认存储在 tb_partition_employees 数据表的 p0 分区中的记录数与 tb_swap 数据表的记录数。

```
MariaDB> SELECT COUNT(*) FROM tb_partition_employees PARTITION(p0);
+----------+
| COUNT(*) |
+----------+
|   164797 |
+----------+

MariaDB> SELECT COUNT(*) FROM tb_swap;
+----------+
| COUNT(*) |
+----------+
|        1 |
+----------+
```

下面将 tb_partition_employees 数据表的 p0 分区的表空间与 tb_swap 数据表的表空间相互切换。

```
MariaDB> ALTER TABLE tb_partition_employees EXCHANGE PARTITION p0 WITH TABLE tb_swap;
Query OK, 0 rows affected (0.46 sec)
```

从上述执行结果看，表空间切换时没有发生任何问题，大约花费 0.46 秒。该过程中，由于 tb_partition_employees 数据表与 tb_swap 数据表拥有元数据锁，所以会阻止类似 INSERT 或 UPDATE 的 DML 语句。数据表空间切换过程中不会对数据表的记录进行实际物理复制，所以处理速度会非常快。但查询执行频繁的数据表中，进行表空间切换时一定要格外小心。

为了查看切换结果，再次执行下列查询，以查看记录条数。

```
MariaDB> SELECT COUNT(*) FROM tb_partition_employees PARTITION(p0);
+----------+
| COUNT(*) |
+----------+
|        1 |
+----------+

MariaDB> SELECT COUNT(*) FROM tb_swap;
+----------+
| COUNT(*) |
+----------+
|   164797 |
+----------+
```

从查询结果可以看到，tb_partition_employees 数据表的 p0 分区与 tb_swap 数据表的记录条数正好与切换之前的条数对调。由于数据表空间发生交换，各数据表的记录也会发生交换。数据表拥有的记录不符合交换分区的分区规则（分区范围）时，就会发生如下错误，从而导致处理失败，请各位注意。

```
MariaDB> ALTER TABLE tb_partition_employees EXCHANGE PARTITION p1 WITH TABLE tb_swap;
ERROR 1737 (HY000): Found a row that does not match the partition
```

7.5 备份 MariaDB MySQL

RDBMS 中，备份是非常重要的。虽然可以通过复制将同样的数据分别保存到主服务器与从服务器，但备份仍然是必须的。借助备份不仅可以解决系统故障，还可以恢复用户误删的数据。2012 年之前主要使用的是 MySQL 5.0 与 5.1 版本，名为 mysqldump 的逻辑数据转储工具是 MySQL 服务器唯一的免费备份工具。当然也可以通过 Linux 的 LVM（Logical Volume Manager）或 DRBD 这类磁盘块复制进行物理文件备份，但 LVM 与 DRBD 和 MySQL 服务器一起使用时，在性能与稳定性方面表现得不够好。

如果各位使用过 mysqldump 工具就会知道，进行大量数据文件备份时会面临相当多的问题。备份本身属于逻辑备份，对 MySQL 服务器的性能会产生很大影响，耗费的时间也比较长。

但 mysqldump 备份最大的问题在于数据恢复过程。如果曾使用 mysqldump 工具对 100 GB 的数据文件进行备份和数据恢复，那么大家肯定能明白其中的苦楚。服务处于瘫痪状态，但需要 10~12 小时才能恢复数据，浪费这么多时间真是件令人抓狂的事情。

为了解决上述这些问题，Percona 公司开发了 XtraBackup 备份工具。在 Oracle 企业级备份工具与 Percona 的 XtraBackup 之间，当然推荐各位使用 XtraBackup。这并不是因为 XtraBackup 是免费软件，而是因为 XtraBackup 拥有比 Oracle 企业级备份工具更多的功能，而且提供了大量能够帮助管理员解决各种意外问题的设置选项。Percona 公司长期从事 MySQL 技术支持服务，并运营着大量系统，其细腻缜密的特质很好地融合到 XtraBackup 中。XtraBackup 并非只用于 MariaDB 或 Percona 的 PerconaServer 中进行备份，也可以用于 Oracle 的 MySQL 社区版或企业版的备份作业。

此外令人高兴的是，MySQL 5.6 与 MariaDB 10.0 开始添加了在远程服务器中对二进制日志进行备份的功能。二进制日志不仅可以用于复制，也可以进行 PIT（Point In Time）恢复。下面将详细讲解 MySQL 5.6 与 MariaDB 10.0 中新增的二进制日志备份方法以及 XtraBackup 的功能。

7.5.1 二进制日志远程备份

仅在本地服务器对二进制日志文件进行备份通常并非难事。但为了安全起见，备份还要保存到远程服务器。因此，进行二进制日志备份时，需要先在本地服务器进行复制，然后再发送到远程服务器。该过程中，虽然并不需要很难的技术，但其中包含着许多可能引起问题或错误的因素。因此，二进制日志文件的备份功能实现之后，具体应用时经常出现无法正常运行的情形。

从 MySQL 5.6 或 MariaDB 10.0 开始，管理员可以使用 mysqlbinlog 实用工具程序将远程服务器中的二进制日志实时复制到本地计算机。mysqlbinlog 实用工具可以将以二进制形式存储的二进制日志文件解码为文本文件，以便人们阅读，也可以在 MySQL 服务器中再次运行二进制日志文件的内容。从 MySQL 5.6 或 MariaDB 10.0 开始，在 mysqlbinlog 原有功能的基础上，添加了远程服务器二进制日志备份功能，并且从 MySQL 5.6 开始公开了 Replication API。mysqlbinlog 的实时二进制日志备份功能并非将远程服务器的文件复制过来，其运行方式是通过 MySQL 5.6 的 Replication API 实时获取二进制日志事件。其实，mysqlbinlog 的二进制日志备份功能就像一个 MySQL 从服务器一样。因此，使用 mysqlbinlog 二进制日志备份时，不需要其他远程服务器的 SSH 或操作系统级别的用户账号。

使用 mysqlbinlog 进行二进制日志备份时，需要使用下列选项。

--read-from-remote-server

该选项是 mysqlbinlog 实用工具的核心选项，用于备份远程服务器中的二进制日志。若不使用该选项，mysqlbinlog 就会像以前一样查找本地磁盘上的二进制日志，并将其转换为文本文件。使用 mysqlbinlog 实用工具和 --read-from-remote-server 选项时，必须一并使用 --host、--user、--password、--port 选项，这些选项含有连接远程服务器必需的信息。

--raw

使用该选项时，mysqlbinlog 实用工具会将获取的二进制日志（二进制形式）原原本本记录到磁盘。若不使用该选项，mysqlbinlog 实用工具就会生成变换为文本形式的二进制日志。

--stop-never

使用 mysqlbinlog 实用工具可以只从远程服务器获取指定的几个二进制日志，也会将不断生成的二进制日志文件保存到本地磁盘。使用 --stop-never 选项时，只要远程服务器不关闭或连接未断开，mysqlbinlog 实用工具就会不断复制远程服务器中的二进制日志。

--stop-never-slave-server-id=id

使用 --stop-never 选项后，mysqlbinlog 实用工具工作起来就像一个从服务器。MySQL 复制中，所有服务器必须有固定的 server-id。该选项指定 mysqlbinlog 使用哪个 server-id。若不明确指出，则初始化为最大值 65 535。

--result-file

该选项用于设置保存从远程服务器获取的二进制日志文件的文件名前缀（Prefix）。比如，远程服务器的二进制日志文件为 mysql-bin.000002，将 --result-file 选项设置为 /backup/copied-，那么 mysqlbinlog 就会将备份的二进制日志以 /backup/copied-mysql-bin.000002 文件名进行保存。若想使文件名与远程服务器保持一致，只要将 --result-file 选项设置为 /backup/ 目录即可。请注意，将 --result-file 选项设置为目录时，最后一定要使用目录分隔符“/”。

--to-last-log

该选项与 mysqlbinlog 一起使用时，mysqlbinlog 不仅能够获得指定的二进制日志文件，也能够获得其后生成的所有二进制日志。若仅指定几个二进制日志文件并对其进行备份，则不应使用该选项。此外，使用 --stop-never 选项时也会自动激活 --to-last-log 选项。

以上就是使用 mysqlbinlog 复制远程服务器的二进制日志文件时可以使用的选项。复制连接断开时，MySQL 从服务器会再次自动尝试连接，但 mysqlbinlog 的二进制日志备份功能与 MySQL 从服务器不同，连接断开后不会再次尝试连接与复制。若远程服务器正在进行系统检测，而出现 MySQL 服务器重启或主 MySQL 断开全部连接的情形，mysqlbinlog 将不再复制二进制日志文件。此时，若想使 mysqlbinlog 的远程二进制日志备份自动执行，则需要编写如下所示的脚本。

```
#!/bin/sh

BACKUP_BIN=/mysql5.6/bin/mysqlbinlog
LOCAL_BACKUP_DIR=/backup/binlog/

REMOTE_HOST=192.168.0.1
REMOTE_PORT=3306
REMOTE_USER=replication_user
REMOTE_PASS=replication_pass
```

```
# time to wait before reconnecting after failure
SLEEP_SECONDS=10


## 必要时生成目录
mkdir -p ${LOCAL_BACKUP_DIR}
cd ${LOCAL_BACKUP_DIR}


## 运行 while 循环，连接断开后等待指定时间，再次进行连接
WHILE :
DO
  LAST_FILE='ls -1 ${LOCAL_BACKUP_DIR} | grep -v orig | tail -n 1'
  TIMESTAMP='date +%s'
  FILE_SIZE=$(stat -c%s "$LASTFILE")

  IF [ ${FILE_SIZE} -gt 0 ]; THEN
    echo "重命名未完成的备份文件。"
    mv ${LAST_FILE} ${LAST_FILE}.orig_${TIMESTAMP}
  FI

  touch ${LAST_FILE}
  echo "从二进制日志文件开始继续备份。"
  ${BACKUP_BIN} --raw --read-from-remote-server --stop-never --host ${REMOTE_HOST} \
        --port ${REMOTE_PORT} --user ${REMOTE_USER} --password ${REMOTE_
PASS} ${LAST_FILE}

  echo "mysqlbinlog 终止。返回代码：$?"
  echo "${SLEEP_SECONDS} 秒后再次连接并继续备份。"
  sleep ${SLEEP_SECONDS}
DONE
```

上述简单脚本程序中运行着 while 循环，连接断开后，会先等待一段指定时间（10 秒），然后再次连接远程服务器，并从断开的文件处继续备份。虽然这段脚本程序不够完整，但包含了所有核心功能，各位使用时再根据所需服务进行补充调整即可。

注意 当然，有些用户可能另外编写了脚本与程序以对远程服务器的二进制日志进行备份。需要注意，若这些脚本或程序是单独启动的，不随 MySQL 服务器或 MariaDB 服务器一起启动，那么它们将无法对 MySQL 服务器当前正在写的二进制日志文件进行备份。因此，总是在 MySQL 服务器全部写完并转换为文件之后才可能进行备份。根据服务特性，写完 1 个二进制日志文件耗费的时间不同，可能需要 1~2 天。也就是说，备份周期可能变得非常长。

直接连接 MySQL 服务器后，每当发生二进制日志事件时，mysqlbinlog 实用工具就会实时备份日志。与普通复制类似，主服务器中发生事件后，一般都会在 0.5~1 秒之内进行备份。

写作本书时，MySQL 5.6.15 已经发布。虽然它是稳定版本，但从现有版本直接升级到 MySQL 5.6 版本会有一定难度。由于该功能安装在不同于 MySQL 与 MariaDB 的服务器中，所以为了对正处于服务状态的 MySQL 5.5 或 MariaDB 5.5 服务器的二进制日志进行实时备份，也可以只使用 MySQL 5.6 或 MariaDB 10.0 的 mysqlbinlog 实用工具。

像这样，经过实时备份得到的二进制日志也可以用于 XtraBackup 的 PIT 恢复。若 MySQL 服务器保存着非常重要的数据，则可以考虑对二进制日志进行实时备份。

7.5.2 XtraBackup 介绍

XtraBackup 是 Percona 公司开发的物理级开源备份工具。XtraBackup 拥有 MySQL 企业级备份工具提供的所有功能，还提供了许多更有用的管理功能。XtraBackup 工具不仅可以备份 PerconaServer 中的数据，还可以备份 MariaDB 与 MySQL 服务器中的数据。可以从 Percona 网站（http://www.percona.com/software/percona-xtrabackup）下载 XtraBackup 程序。

1. XtraBackup 包结构

XtraBackup 由 innobackupex（用于启动备份的 Perl 脚本）、xtrabackup[①]（对 InnoDB 或 XtraDB 进行备份）二进制程序以及几个实用工具程序组成。下面简单介绍各应用程序的用法与作用。

innobackupex

innobackupex 是用 Perl 语言编写的脚本，负责在 XtraBackup 中调用其他备份实用工具进行整体备份。也就是说，它充当实际执行备份功能的程序的包装器（Wrapper）。此外，innobackupex 脚本也可以直接对使用（除 InnoDB 或 XtraDB 之外的）存储引擎的数据表进行备份。因此，在 MySQL 与 MariaDB 服务器中实际备份数据时，总要使用 innobackupex 脚本。innobackupex 脚本拥有相当多的选项，本书只讲解其中一些常用功能。若需要更多功能，请查看使用手册。

xtrabackup

xtrabackup 程序由 C 语言编写和编译，不保证在各操作系统下都兼容，负责对使用 InnoDB 或 XtraDB 存储引擎的数据表进行实时备份，主要收集并存档 InnoDB 或 XtraDB 的重做日志，或者处理对数据文件的复制。xtrabackup 命令需要 --backup 与 --prepare 选项，一般由 innobackupex 所需选项进行驱动，所以 xtrabackup 程序的使用方法显得并不怎么重要。

xbstream

xbstream 负责将 XtraBackup 创建的备份转换为流。备份处理为流后，即可使用 Unix 的管道（"|"）命令重定向为第三个应用程序的标准输入，或者传送到远程服务器，所以 xbstream 是相当重要的程序。7.5.3 节将讲解可以在 xbstream 程序中使用的选项。

---

① 此处大写字母开头的 XtraBackup 指备份包本身，而小写形式的 xtrabackup 则指包含于 XtraBackup 包的二进制程序的名称。

xbcrypt

使用 xbstream 程序将 XtraBackup 的结果转换为流后，重定向至 xbcrypt 程序的标准输入，然后 xbcrypt 就会从几个加密算法中选择一个，创建经过加密的备份文件。7.5.3 节将详细讲解 xbcrypt 程序。

以上就是 XtraBackup 包的组成程序及各程序的作用，下面继续讲解 XtraBackup 的工作原理。

2. XtraBackup 的工作原理

XtraBackup 的基本原理是使用 InnoDB 存储引擎的自动故障恢复（Crash-Recovery）功能。先忽略其他存储引擎，下面简单介绍使用 XtraBackup 对 InnoDB 或 XtraDB 存储引擎数据表进行备份并恢复的过程。图 7-8~ 图 7-10 按照时间顺序描述了从 00:00 开始备份到中间备份的过程（包含开始备份后、中间时间点，以及最后备份完成时的备份进展情况），以及备份时处理进入 MariaDB 的 INSERT、UPDATE、DELETE 查询的情形。

如图 7-8 所示，假设 00:10 时无任何数据变更，即 MariaDB 的 XtraDB 数据表的磁盘数据文件完全未改动。那么，XtraBackup 只要原样复制 MariaDB 的数据文件即可。

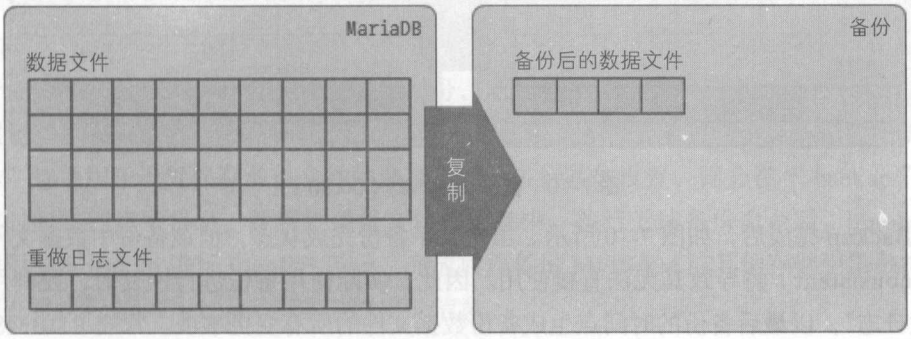


图 7-8 备份开始后（00:10）

图 7-9 表示 00:20 前后的备份进展。假设此时用户执行了 UPDATE 语句，数据文件的前部与中间部分就会更改。但由于 XtraBackup 已经复制了数据文件的前部，所以数据文件前面的变更内容不会应用到备份。然而，此时 XtraBackup 会在重做日志中追踪有关 InnoDB 或 XtrDB 数据变动的项，并只将变更部分收集到另外的备份区域。图 7-9 中的这块备份区域称为“被备份的重做日志”，有时也称为“被存档的日志”。

图 7-10 表示备份结束，从图中可以看到，左侧 MariaDB 服务器的数据文件与右侧备份文件的内容有很大不同。这是因为，如果 MariaDB 服务器一直有变更事项进入，那么复制备份数据文件时就没有快照（Snapshot）功能。复制备份时，由于是未通过特定时间点的快照进行的，所以各备份区域在每个备份时间点都不同，这将导致状态不一致。此外，由于“被备份的重做日志”在 MariaDB 的重做日志记录之后进行内容追踪，所以可以只将更改（Append）过的内容收集留存。

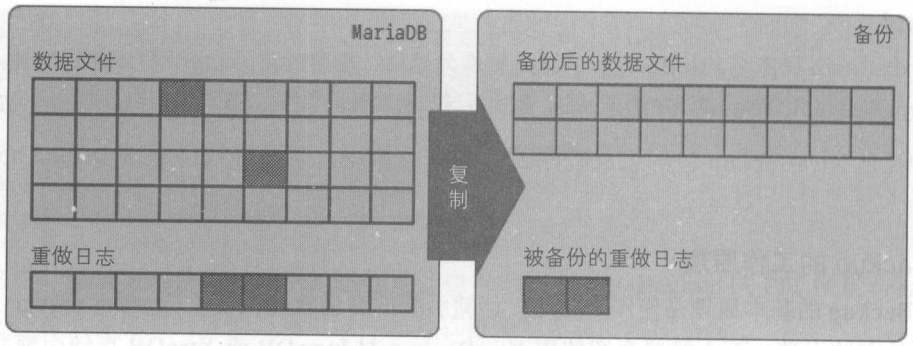


图 7-9 备份开始后数据变更时刻（00:20）

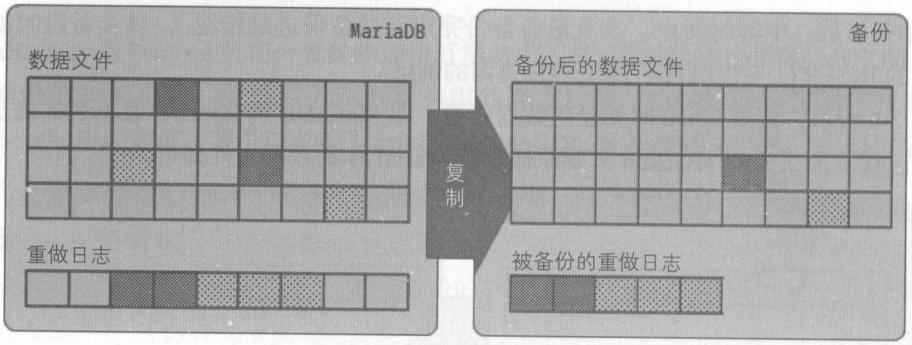


图 7-10 备份结束时刻（00:30）

XtraBackup 完成后，如图 7-10 所示。虽然处于备份完成状态，但该备份中数据文件的不一致性（inconsistent）将导致其无法直接使用。因此，实际使用备份进行恢复时，要利用“被备份的重做日志”，以最后备份的时间点生成备份数据文件的所有变更事项。常规 RDBMS 将该过程称为“前滚”（Roll forward），与 InnoDB 与 XtraDB 存储引擎的故障恢复过程一样。XtraBackup 将该过程称为“apply-log”，与 MySQL 或 MariaDB 服务器启动过程一样，XtraBackup 也经历相同过程，同时进行恢复操作。该过程完成后，最终备份数据文件如图 7-10 左侧“数据文件”所示。

以上就是 InnoDB 或 XtraDB 这类支持事务的存储引擎。令人遗憾的是，MariaDB 服务器中除了这两个存储引擎外，并不支持提供服务过程中进行实时物理备份的工具。在其他存储引擎中进行备份时，需要先终止执行所有查询，然后再复制物理文件。XtraBackup 中使用 FLUSH TABLES WITH READ LOCK 命令（当然 FLUSH TABLES WITH READ LOCK 命令也有其他用途）终止执行所有查询。若所有用户数据都是 InnoDB 或 XtraDB 数据表，则由于其他都是小数据表（比如 MyISAM 数据表），所以需要全局锁的时间非常短暂，各位不必太担心。此外，由于存在于 MariaDB 或 MySQL 服务器的 MyISAM 数据表大部分几乎没有变动，所以备份时也可以不加锁。

3. XtraBackup 的基本用法

下面通过简单示例学习使用 XtraBackup 工具进行备份以及恢复备份的方法。使用 XtraBackup 的整个过程大致分为如下三个步骤，括号中的内容表示执行 innobackupex 程序时要使用的选项。

```
备份（--bakup）=> 应用日志（--apply-log）=> 恢复（--copy-back）
```

(1) 备份

备份时要运行 innobackupex 脚本，下面这些选项通常都是必需的。

```
innobackupex \
  --host="127.0.0.1" \
  --user="backup_user" \
  --password='backup_user_password' \
  --port=3306 \
  --defaults-file="/etc/my.cnf" \
  --backup  /backup/
```

--default-file 选项用于设置要备份的 MariaDB 服务器使用的 my.cnf 文件路径；--backup 选项表示运行 innobackupex 脚本的目的是进行备份，此处可不明示；执行 innobackupex 脚本时，若均为明示 --apply-log、--copy-back、--backup 选项，则默认使用 --backup 选项。最后的“/backup/”选项用于设置保存备份文件的路径。若不进行其他设置，则会在“/backup/”目录下使用当前时间戳创建子目录，然后将备份文件存储其中。执行上述备份命令后，innobackupex 脚本会调用 xtrabackup 进行 InnoDB 备份，完成后再备份 MyISAM 或其他存储引擎数据表的数据。下面查看执行 innobackupex 脚本时输出的信息。

下列信息是执行 innobackupex 时输出的有关许可证、MariaDB 服务器版本以及连接测试等的信息。

```
InnoDB Backup Utility v1.5.1-xtrabackup; Copyright 2003, 2009 Innobase Oy
and Percona LLC and/or its affiliates 2009-2013.  All Rights Reserved.

This software is published under
the GNU GENERAL PUBLIC LICENSE Version 2, June 1991.

Get the latest version of Percona XtraBackup, documentation, and help resources:
http://www.percona.com/xb/p

140206 12:13:28  innobackupex: Connecting to MySQL server with DSN '...
140206 12:13:28  innobackupex: Connected to MySQL server
140206 12:13:28  innobackupex: Executing a version check against the server...
140206 12:13:28  innobackupex: Done.
IMPORTANT: Please check that the backup run completes successfully.
```

```
        At the end of a successful backup run innobackupex
        prints "completed OK!".

innobackupex: Using mysql server version 10.0.7-MariaDB-log
```

然后创建保存备份的目录，并显示目录路径。

```
innobackupex: Using mysql server version 10.0.7-MariaDB-log
innobackupex: Created backup directory /backup/2014-02-07_22-35-43
```

下面执行备份，innobackupex 脚本在内部启动 xtrabackup 程序。此时，根据备份目标不同，MySQL 服务器的版本启动的 xtrabackup 程序版本也应不同。XtraBackup 完全采用 InnoDB 或 XtraDB 存储引擎的故障恢复过程，该过程会随版本的不同而有所差别。因此，针对不同版本的 MySQL，xtrabackup 程序自然也存在不同版本。从上述示例可以看到，由于备份目标服务器是 MariaDB 10.0，所以执行了 xtrabackup_56 程序。从后面可以看到，xtrabackup_56 程序读取了 MariaDB 服务器的设置信息。

```
140207 22:35:43  innobackupex: Starting ibbackup with command: xtrabackup_56
  --defaults-file="/etc/my.cnf"  --defaults-group="mysqld" --backup -- suspend-
at-end --target-dir= /backup/2014-02-07_22-35-43 --tmpdir=/tmp --throttle=5
innobackupex: Waiting for ibbackup (pid=32516) to suspend
innobackupex: Suspend file '/backup/2014-02-07_22-35-43/xtrabackup_suspended_2'

xtrabackup_56 version 2.1.7 for MySQL server 5.6.15 Linux (x86_64) (revision id: 721)
xtrabackup: uses posix_fadvise( ).
xtrabackup: cd to /data/mysql
xtrabackup: using the following InnoDB configuration:
xtrabackup:   innodb_data_home_dir = /mysql
xtrabackup:   innodb_data_file_path = ibdata1:100M:autoextend
xtrabackup:   innodb_log_group_home_dir = /mysql/log
xtrabackup:   innodb_log_files_in_group = 2
xtrabackup:   innodb_log_file_size = 1572864000
2014-02-07 22:35:43 2b84e7d230f0 InnoDB: Using Linux native AIO
```

上述信息中，需要引起重视的部分是“uses posix_fadvise()”。其中，posix_fadvise() 函数为 Linux 的内核函数。使用该选项后，读写文件的内容不会保存到 Linux 操作系统的缓存，而可以直接清除。假如不使用该选项备份 100 GB 的数据文件，Linux 操作系统的缓存就会填满一次读取所有剩余内存后不再引用的数据。虽然也有双缓冲的问题，但 Linux 缓存有时会将 MySQL 服务器的 InnoDB 缓冲池换出到磁盘。为了解决这些问题，XtraBackup 执行过程中读写的文件会调用 posix_fadvise() 函数。

接下来启动 2 个线程，一个线程用于备份 innoDB 数据表的数据文件，另一个线程用于跟踪 InnoDB 的重做日志，将变更内容存档到另外的文件。

```
>> log scanned up to (742923074)
InnoDB: Allocated tablespace 764, old maximum was 0
[01] Copying /mysql/data/ibdata1 to /backup/2014-02-07_22-35-43/ibdata1
>> log scanned up to (742923074)
...
>> log scanned up to (742923074)
[01] Copying ./test/test.ibd to /backup/2014-02-07_22-35-43/test/test.ibd
[01]        ...done
[01] Copying ./test/test1.ibd to /backup/2014-02-07_22-35-43/test/test1.ibd
[01]        ...done
>> log scanned up to (742923074)
xtrabackup: Creating suspend file '/backup/2014-02-07_22-35-43/xtrabackup_
suspended_2' with pid '32516'
```

像这样，InnoDB 数据表备份完成后，innobackupex 程序就开始复制不使用 InnoDB 或 XtraDB 存储引擎的数据表。innobackupex 程序复制数据文件期间，xtrabackup 程序会一直等待，同时追踪 InnoDB 的重做日志的变更部分，并收集为归档文件。最终，innobackupex 程序完成数据文件备份，xtrabackup 程序终止收集重做日志，然后将收集到的重做日志的位置（LSN，Log Sequence Number）输出为信息。最后的“xtrabackup:Transaction log of lsn(742879760) to (742879760) was copied”信息表示本次备份的起始 ISN 为 742879760，备份到 LSN 742879760 时间点完成备份。此处最后的 LSN 时间点在执行增量备份时也会用到，所以也将其记录到备份目录中的 xtrabackup_checkpoints 文件。

```
140207 22:36:07  innobackupex: Continuing after ibbackup has suspended

140207 22:36:07  innobackupex: Starting to backup non-InnoDB tables and files
innobackupex: in subdirectories of '/data/mysql'
innobackupex: Backing up files '${MYSQL_HOME}/performance_schema/*.{frm,isl,
MYD,MYI,MAD,MAI,MRG,TRG,TRN,ARM,ARZ,CSM,CSV,opt,par}' (18 files)
innobackupex: Backing up files '${MYSQL_HOME}/test/*.{frm,isl,MYD,MYI,MAD,MAI,MRG,
TRG,TRN,ARM,ARZ,CSM,CSV,opt,par}' (22 files)
innobackupex: Backing up files '${MYSQL_HOME}/mysql/*.{frm,isl,MYD,MYI,MAD,MAI,MRG,
TRG,TRN,ARM,ARZ,CSM,CSV,opt,par}' (79 files)
...
140207 22:36:08  innobackupex: Finished backing up non-InnoDB tables and files

140207 22:36:08  innobackupex: Waiting for log copying to finish

xtrabackup: The latest check point (for incremental): '742923074'
xtrabackup: Stopping log copying thread.
.>> log scanned up to (742923074)
```

```
xtrabackup: Creating suspend file '/backup/2014-02-07_22-35-43/xtrabackup_log_copied'
with pid '32516'
xtrabackup: Transaction log of lsn (742923074) to (742923074) was copied.
```

至此完成全部备份工作，查看位于输出信息末尾的“completed OK!”信息，可以判断备份是否正常结束。

```
innobackupex: Backup created in directory '/backup/2014-02-07_22-35-43'
140207 22:36:09 innobackupex: Connection to database server closed
140207 22:36:09 innobackupex: completed OK!
```

备份完成后，/backup/2014-02-06_12-13-28 目录中会出现如下内容。

```
shell> ls -alh /backup/2014-02-06_12-13-28/
合计 101M
drwxr-xr-x 10 root root 4.0K  2月  7 22:40 .
drwxr-xr-x  3 root root 4.0K  2月  7 22:40 ..
-rw-r--r--  1 root root  360  2月  7 22:40 backup-my.cnf
-rw-r-----  1 root root 100M  2月  7 22:40 ibdata1
drwxr-xr-x  2 root root 4.0K  2月  7 22:40 mysql
drwxr-xr-x  2 root root 4.0K  2月  7 22:40 performance_schema
drwx------  2 root root 4.0K  2月  7 22:40 test
-rw-r--r--  1 root root   13  2月  7 22:40 xtrabackup_binary
-rw-r-----  1 root root   93  2月  7 22:40 xtrabackup_checkpoints
-rw-r-----  1 root root 2.5K  2月  7 22:40 xtrabackup_logfile
```

备份目录下除了数据文件外，还有如下存储元信息的文件。

xtrabackup_checkpoints

包含用于区分是完全备份还是增量备份的信息，还存储着备份的最后 LSN。

xtrabackup_binlog_info

保存备份完成时二进制日志文件的名称与位置。

xtrabackup_slave_info

innobackupex 使用 --slave-info 选项执行时才会生成该文件，它保存着进行备份的 MySQL 服务器中主 MySQL 服务器的二进制日志文件的名称与位置，即保存着从服务器中执行 SHOW SLAVE STATUS 命令时要显示的信息。

backup-my.cnf

备份目标 MariaDB 服务器的 my.cnf 设置文件中仅保存备份与恢复所需的几种设置内容（innodb_data_file_path、innodb_log_flies_in_group、innodb_log_file_size、innodb_fast_checksum、innodb_page_size、innodb_log_block_size）。

xtrabackup_binary

保存用于备份 InnoDB 或 XtraDB 数据表的 xtrabackup 程序的名称。根据备份时所使用的 xtrabackup 程序版本，恢复时也要使用同版本的 xtrabackup 程序。

备份目录下的 xtrabackup_checkpints 文件内容如下所示。以后恢复或增量备份时也会用到 backup_type 与 to_lsn 等信息，所以不能修改或删除。

```
backup_type = full-backuped
from_lsn = 0
to_lsn = 742923074
last_lsn = 742923074
compact = 0
```

(2) 应用日志

使用 XtraBackup 进行备份时，复制备份数据文件与备份的过程中，就会生成收集的重做日志文件，如图 7-10 所示。应用日志指的是 InnoDB 或 XtraDB 的故障恢复过程。如前所述，由于备份文件的所有块（包）不是特定时间点的一致备份，所以应用日志是指将收集到的重做日志应用到备份数据文件。如果每次只进行完全备份，那么应用日志可以在备份完成之后进行，也可以在恢复之前进行。但在增量备份中，最好在恢复之前就应用日志。

若要应用日志，只需要在执行 innobackupex 脚本的同时使用 --apply-log 选项。

```
shell> innobackupex --apply-log /backup/2014-02-07_22-35-43
```

与备份时一样，先输出版本相关信息，然后显示 innobackupex 使用哪个选项启动 xtrabackup 程序。此外，xtrabackup 引用备份设置文件，在画面中显示重做日志与数据目录。通过 innobackupex 脚本执行的 xtrabackup 程序接着执行 InnoDB/XtraDB 故障恢复过程，恢复过程中输出的所有信息都以“InnoDB:”开头。以“InnoDB:”开头的信息与 InnoDB 引擎启动时向 MariaDB 服务器的错误日志输出的引擎初始化内容是一样的，其实，xtrabackup 进程启动了 InnoDB 存储引擎，但此时启动的 InnoDB 只引用备份文件。也就是说，即使在当前工作的服务器中安装其他 MySQL 服务器或 MySQL 服务器，并且服务正在执行，它们相互之间也不会产生任何影响。重做日志应用过程结束后，显示“InnoDB:Shutdown completed”信息并退出。

```
140207 22:38:27  innobackupex: Starting ibbackup with command: xtrabackup_56
--defaults-file="/backup/2014-02-07_22-35-43/backup-my.cnf"  --defaults-group=
"mysqld" --prepare --target-dir=/backup/2014-02-07_22-35-43 --tmpdir=/tmp

xtrabackup_56 version 2.1.7 for MySQL server 5.6.15 Linux (x86_64) (revision id: 721)
xtrabackup: cd to /backup/2014-02-07_22-35-43
xtrabackup: This target seems to be not prepared yet.
xtrabackup: xtrabackup_logfile detected: size=2097152, start_lsn=(742923074)
xtrabackup: using the following InnoDB configuration for recovery:
```

```
xtrabackup:   innodb_data_home_dir = ./
xtrabackup:   innodb_data_file_path = ibdata1:100M:autoextend
xtrabackup:   innodb_log_group_home_dir = ./
xtrabackup:   innodb_log_files_in_group = 1
xtrabackup:   innodb_log_file_size = 2097152
xtrabackup: using the following InnoDB configuration for recovery:
xtrabackup:   innodb_data_home_dir = ./
xtrabackup:   innodb_data_file_path = ibdata1:100M:autoextend
xtrabackup:   innodb_log_group_home_dir = ./
xtrabackup:   innodb_log_files_in_group = 1
xtrabackup:   innodb_log_file_size = 2097152
xtrabackup: Starting InnoDB instance for recovery.
xtrabackup: Using 104857600 bytes for buffer pool (set by --use-memory parameter)
InnoDB: The InnoDB memory heap is disabled
InnoDB: Mutexes and rw_locks use GCC atomic builtins
InnoDB: Compressed tables use zlib 1.2.3
InnoDB: Using CPU crc32 instructions
InnoDB: Initializing buffer pool, size = 100.0M
InnoDB: Completed initialization of buffer pool
InnoDB: Highest supported file format is Barracuda.
InnoDB: The log sequence numbers 742895206 and 742895206 in ibdata files do
not match the log sequence number 742923074 in the ib_logfiles!
InnoDB: Database was not shutdown normally!
InnoDB: Starting crash recovery.
InnoDB: Reading tablespace information from the .ibd files...
InnoDB: Restoring possible half-written data pages
InnoDB: from the doublewrite buffer...
InnoDB: In a MySQL replication slave the last master binlog file
InnoDB: position 0 5582949, file name mysql-binary.000002
InnoDB: Last MySQL binlog file position 0 627, file name /log/mysql-log/
mysql-binary.000002
InnoDB: 128 rollback segment(s) are active.
InnoDB: Waiting for purge to start
InnoDB: 5.6.15 started; log sequence number 742923074

[notice (again)]
  If you use binary log and don't use any hack of group commit,
  the binary log position seems to be:
InnoDB: Last MySQL binlog file position 0 627, file name /mysql/log/mysql-binary.000002

xtrabackup: starting shutdown with innodb_fast_shutdown = 1
InnoDB: FTS optimize thread exiting.
```

```
InnoDB: Starting shutdown...
InnoDB: Shutdown completed; log sequence number 742924828
```

有时需要额外作业，xtrabackup 程序会再次执行 InnoDB 的恢复过程。同时，为了恢复操作也会提前生成重做日志文件。

```
140207 22:38:29  innobackupex: Restarting xtrabackup with command: xtrabackup_56
  --defaults-file="/backup/2014-02-07_22-35-43/backup-my.cnf"  --defaults-group=
"mysqld" --prepare --target-dir=/backup/2014-02-07_22-35-43 --tmpdir=/tmp
for creating ib_logfile*

xtrabackup_56 version 2.1.7 for MySQL server 5.6.15 Linux (x86_64) (revision id: 721)
xtrabackup: cd to /backup/2014-02-07_22-35-43
...
InnoDB: The InnoDB memory heap is disabled
InnoDB: Mutexes and rw_locks use GCC atomic builtins
InnoDB: Compressed tables use zlib 1.2.3
InnoDB: Using CPU crc32 instructions
InnoDB: Initializing buffer pool, size = 100.0M
InnoDB: Completed initialization of buffer pool
InnoDB: Setting log file ./ib_logfile101 size to 1500 MB
InnoDB: Progress in MB: 100 200 300 400 500 600 700 800 900 1000 1100 1200 1300 1400 1500
InnoDB: Setting log file ./ib_logfile1 size to 1500 MB
InnoDB: Progress in MB: 100 200 300 400 500 600 700 800 900 1000 1100 1200 1300 1400 1500
InnoDB: Renaming log file ./ib_logfile101 to ./ib_logfile0
InnoDB: New log files created, LSN=742924828
InnoDB: Highest supported file format is Barracuda.
InnoDB: 128 rollback segment(s) are active.
InnoDB: Waiting for purge to start
InnoDB: 5.6.15 started; log sequence number 742925324

[notice (again)]
  If you use binary log and don't use any hack of group commit,
  the binary log position seems to be:
InnoDB: Last MySQL binlog file position 0 627, file name /mysql/log/mysql-binary.000002

xtrabackup: starting shutdown with innodb_fast_shutdown = 1
InnoDB: FTS optimize thread exiting.
InnoDB: Starting shutdown...
InnoDB: Shutdown completed; log sequence number 742926978
```

日志应用过程全部完成后，innobackupex 脚本将显示信息“innobackupex:completed OK!”，表明整个过程全部完成，如下所示。

```
140207 22:38:57  innobackupex: completed OK!
```

日志应用过程中显示的内容相当多，但依然要将这些内容全部收录。实际应用日志与恢复备份时需要多种信息，这些信息在恢复备份时就会这样输出显示。各过程中带下划线的部分是比较重要的，请各位注意。

(3) 恢复

日志应用全部完成，接下来只需要将备份内容原样复制到 MySQL 或 MariaDB 服务器的数据目录。其实，也可以将备份目录全部复制到 MySQL 或 MariaDB 服务器的数据目录，但 MariaDB 服务器的数据目录就会出现不必要的文件。因此，若可能，请使用 innobackupex 脚本的 --copy-back 命令。运行 innobackupex 命令前，MySQL 或 MariaDB 服务器必须终止，并且必须清空数据目录与重做日志目录。此外，执行 copy-back 前，MySQL 或 MariaDB 服务器必须处于可启动状态。innobackupex 脚本的最后参数必须明确指出保存备份的目录。

```
shell> innobackup --copy-back /backup/2014-02-07_22-35-43
```

执行 copy-back 作业时，MySQL 或 MariaDB 服务器运行时所需的所有设置文件、数据文件，以及 apply-log 过程中事先生成的重做日志文件都会复制到 MySQL 服务器或 MariaDB 的各个目录。首先复制各数据库的数据文件与数据表元数据文件（.FRM）。

```
InnoDB Backup Utility v1.5.1-xtrabackup; Copyright 2003, 2009 Innobase Oy
and Percona LLC and/or its affiliates 2009-2013.  All Rights Reserved.

This software is published under
the GNU GENERAL PUBLIC LICENSE Version 2, June 1991.

Get the latest version of Percona XtraBackup, documentation, and help resources:
http://www.percona.com/xb/p

IMPORTANT: Please check that the copy-back run completes successfully.
           At the end of a successful copy-back run innobackupex
           prints "completed OK!".

Warning: xtrabackup: ignoring option '--innodb_adaptive_hash_index' due to invalid value 'OFF'
innobackupex: Starting to copy files in '/backup/2014-02-07_22-35-43'
innobackupex: back to original data directory '/mysq/data'
innobackupex: Copying '/backup/2014-02-07_22-35-43/xtrabackup_binlog_pos_innodb'
to '/mysq/data/xtrabackup_binlog_pos_innodb'
innobackupex: Creating directory '/mysq/data/performance_schema'
...
innobackupex: Creating directory '/mysq/data/test'
innobackupex: Copying '/backup/2014-02-07_22-35-43/test/test.frm' to '/mysq/data/
```

```
test/test.frm'
innobackupex: Copying '/backup/2014-02-07_22-35-43/test/test1.ibd' to '/mysq/data/
test/test1.ibd'
innobackupex: Creating directory '/mysq/data/mysql'
...
innobackupex: Copying '/backup/2014-02-07_22-35-43/mysql/host.MYD' to '/mysq/data/
mysql/host.MYD'
innobackupex: Copying '/backup/2014-02-07_22-35-43/mysql/time_zone.frm' to '/mysq/
data/mysql/time_zone.frm'
innobackupex: Copying '/backup/2014-02-07_22-35-43/mysql/host.MYI' to '/mysq/data/
mysql/host.MYI'
innobackupex: Copying '/backup/2014-02-07_22-35-43/mysql/time_zone.MYI' to '/mysq/
data/mysql/time_zone.MYI'
innobackupex: Copying '/backup/2014-02-07_22-35-43/mysql/servers.frm' to '/mysq/
data/mysql/servers.frm'
innobackupex: Copying '/backup/2014-02-07_22-35-43/mysql/proc.frm' to '/mysq/data/
mysql/proc.frm'
innobackupex: Copying '/backup/2014-02-07_22-35-43/mysql/db.opt' to '/mysq/data/
mysql/db.opt'
innobackupex: Copying '/backup/2014-02-07_22-35-43/mysql/func.MYD' to '/mysq/data/
mysql/func.MYD'
...
```

数据文件复制完成后，将 InnoDB 或 XtraDB 的系统表空间 ibdata1, …文件复制到数据目录。

```
innobackupex: Starting to copy InnoDB system tablespace
innobackupex: in '/backup/2014-02-07_22-35-43'
innobackupex: back to original InnoDB data directory '/mysq/data'
innobackupex: Copying '/backup/2014-02-07_22-35-43/ibdata1' to '/mysq/data/ibdata1'
```

若将 Undo 表空间设置为使用另外的表空间，则也会复制 Undo 表空间文件。本示例中不存在另外的 Undo 表空间，所以没有复制任何内容。

```
innobackupex: Starting to copy InnoDB undo tablespaces
innobackupex: in '/backup/2014-02-07_22-35-43'
innobackupex: back to '/mysq/data'
```

最后，将 InnoDB 或 XtraDB 的重做日志复制到日志目录。

```
innobackupex: Starting to copy InnoDB log files
innobackupex: in '/backup/2014-02-07_22-35-43'
innobackupex: back to original InnoDB log directory '/log/innodb-log'
innobackupex: Copying '/backup/2014-02-07_22-35-43/ib_logfile0' to '/log/
innodb-log/ib_logfile0'
innobackupex: Copying '/backup/2014-02-07_22-35-43/ib_logfile1' to '/log/
```

```
innodb-log/ib_logfile1'
innobackupex: Finished copying back files.
```

所有数据文件与重做日志文件复制完成后，输出如下信息，表明全部完成。

```
140207 22:41:35  innobackupex: completed OK!
```

(4) 启动 MySQL/MariaDB 服务器

通过备份文件进行恢复的工作全部完成，下面只要启动 MySQL 或 MariaDB 服务器即可。但启动 MySQL/MariaDB 服务器前，还必须修改数据文件与重做日志文件的权限。若以名为 mysql 的操作系统用户（所谓的 db 操作系统用户组）运行 MySQL 或 MariaDB 服务器，需要修改以下数据文件（/mysql/data）与重做日志（/mysql/log）的权限。

```
shell> chown -R mysql:db /mysql/data /mysql/log
```

接下来，启动 MySQL 或 MariaDB 服务器。若服务器使用恢复数据文件正常启动，则向服务器的错误日志文件输出如下内容。从以下内容可知，XtraDB 从最后备份时间点的重做日志位置开始启动，紧接着显示“CONNECT 存储引擎无法查找到设置文件”信息。这是 MariaDB 10.0 的 CONNECT 存储引擎最近才引入的功能，由于 XtraBackup 尚未准备好备份该设置文件才出现上述状况，所以只需将 XtraDB 无法识别的文件另外进行备份。

```
shell> cat /mysql/data/hostname.err
140207 13:42:22 mysqld_safe Starting mysqld daemon with databases from /mysql/data
140207 13:42:25 [Note] Plugin 'FEEDBACK' is disabled.
2014-02-07 13:42:25 2ac5aac145f0 InnoDB: Warning: Using innodb_additional_
mem_pool_size is DEPRECATED. This option may be removed in future releases,
 together with the option innodb_use_sys_malloc and with the InnoDB's
internal memory allocator.
140207 13:42:25 [Note] InnoDB: The InnoDB memory heap is disabled
140207 13:42:25 [Note] InnoDB: Mutexes and rw_locks use GCC atomic builtins
140207 13:42:25 [Note] InnoDB: Compressed tables use zlib 1.2.3
140207 13:42:25 [Note] InnoDB: Using Linux native AIO
140207 13:42:25 [Note] InnoDB: Using CPU crc32 instructions
140207 13:42:25 [Note] InnoDB: Initializing buffer pool, size = 1.0G
140207 13:42:25 [Note] InnoDB: Completed initialization of buffer pool
140207 13:42:25 [Note] InnoDB: Highest supported file format is Barracuda.
140207 13:42:25 [Note] InnoDB: 128 rollback segment(s) are active.
140207 13:42:25 [Note] InnoDB: Waiting for purge to start
140207 13:42:26 [Note] InnoDB:  Percona XtraDB (http://www.percona.com)
5.6.14-62.0 started; log sequence number 742926978
140207 13:42:26 [Note] CONNECT: connect.ini=/mysq/data/connect.ini
profile file /mysq/data/connect.ini not found
140207 13:42:26 [Note] Server socket created on IP: '::'.
```

```
140207 13:42:26 [Note] Server socket created on IP: '::'.
140207 13:42:26 [Note] Event Scheduler: Loaded 0 events
140207 13:42:26 [Note] /usr/local/mysql/bin/mysqld: ready for connections.
Version: '10.0.7-MariaDB-log'  socket: '/tmp/mysql.sock'  port: 3306  MariaDB Server
```

若要将备份恢复的服务器与其他主 MySQL/MariaDB 进行复制连接，则引用备份目录中存储的二进制日志位置存储文件，从相应位置开始进行连接复制，这样就完成了全部恢复工作。

7.5.3 XtraBackup 的功能

XtraBackup 提供了许多 MySQL 企业版备份工具不支持的有用功能。首先逐个学习 XtraBackup 拥有的功能及特征。

1. 流备份

一般而言，备份工具会将数据库中的数据文件保存到本地磁盘或网络存储系统（NAS）。若有网络存储系统，则可使用其进行备份；若没有网络存储系统，则只能将备份复制到本地磁盘。而在本地磁盘保存备份时，可能引发两个问题：第一个是将备份写入本地磁盘时引起的负荷问题，第二个是磁盘空间问题。将备份存储到本地磁盘是指，读取本地磁盘中的数据文件，然后在本地磁盘的其他空间创建副本。这种本地备份最致命的问题是，服务器出现故障时无法读取磁盘文件而导致无法恢复数据。为了避免该问题，通常先在本地磁盘创建备份副本，然后将备份文件通过网络传送到其他服务器。

XtraBackup 去除了这一多余过程（在本地磁盘创建副本的过程），而是读取本地磁盘数据文件并直接通过网络传送到其他服务器，这称为流备份。在 XtraBackup 中执行流备份时，XtraBackup 不会将备份内容简单地复制到其他目录，而是输出为变形的 tar[①] 或 xbstream 格式。像这样，若采用流方式传送备份，在备份的中间过程可以非常轻松使用多种第三方工具。比如压缩备份时，只要将备份输出流连接到压缩实用工具的输入流即可。另一个典型的例子是加密，与压缩一样，加密实用工具也使用 Unix 管道（“|”），只要将备份输出流连接到加密工具的输入流即可。

使用 XtraBackup 进行流备份时，需要用 xbstream 实用程序，它是 Percona 为 XtraBackup 实用工具开发的基于流的存档实用工具。其实，它做的事情与 tar 类似，tar 或 cpio 这类实用工具无法处理动态生成的文件，而经过改善的 xbstream 可以处理动态文件。xbstream 工具使用的选项与 tar 类似，只支持如下三种典型的选项。

-x

从存档 xbstream 文件逐个解压缩文件。若无 -C 选项，则从当前目录逐个解压缩文件。

① tar 是 Percona 在 Unix 的 tar 实用程序基础上修改而成的，用于 MySQL 备份。现在被 xbstream 工具取代，故几乎不再使用。

-c

将文件逐个加入 1 个 xbstream 存档文件。

-C

与 -x 选项一起使用时，将文件逐个解压缩到指定文件夹。

图 7-11 表示 XtraBackup 流备份的工作原理。XtraBackup 能够同时压缩多个数据文件，一并读取的数据文件通过 xbstream 实用工具归档为 1 个文件，然后以流形式输出。该输出流再经过压缩或加密，最终通过 nc（NetCat）或 rsync 这类实用工具，将文件复制到远程备份存储服务器。

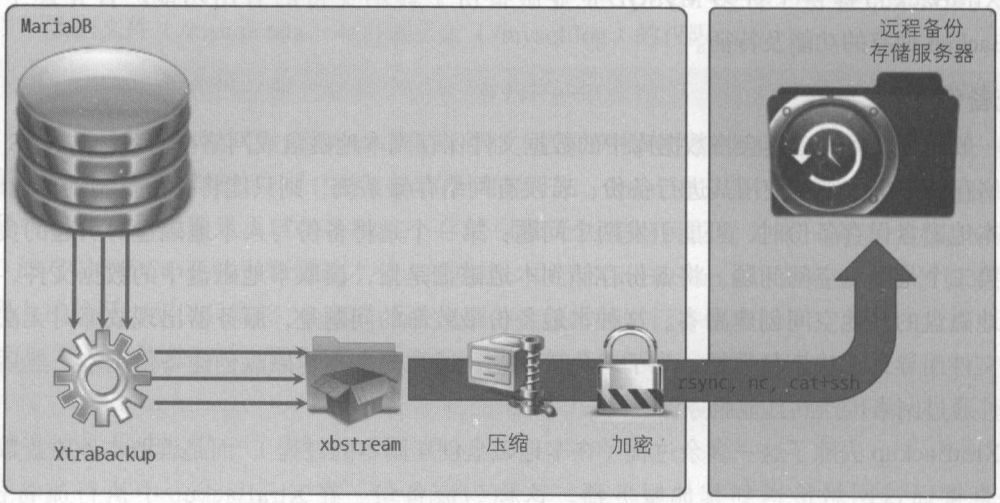


图 7-11 流备份

2. 压缩

现在 XtraBackup 可以对事务日志、数据文件以及元数据进行压缩，但只支持 quicklz 压缩算法。利用 quicklz 算法压缩的文件可以使用 qpress（qpress file archiver，http://www.quicklz.com/）实用工具进行解压缩。

3. 加密

虽然很多人都在努力保证对外提供服务的 MariaDB 服务器的安全，但往往对数据库服务器中备份数据文件的安全疏忽大意。XtraBackup 中添加了对备份数据文件的加密保存功能。Percona 开发了 xbcrypt 实用工具，以加密备份文件，它采用 AES128、AES192、AES256 三种对称加密算法之一。xbcrypt 实用工具既可以对备份文件加密，也可以解密。

xbcrypt 实用工具可以使用如下这些选项。

-d（--decrypt）

对加密文件进行解密。

-i (--input=name)

指出要进行加密或解密的文件。若不明确指出输入文件，则 xbcrypt 实用工具会对来自标准输入流数据进行加密与解密。

-o (--output=name)

指出用于记录加密或解密内容的输出文件。若不明确指出输出文件，则 xbcrypt 工具将加密或解密后的数据发送到标准输出。

-a (--encrypt-alog=name)

指出加密或解密时要使用的加密算法，从 AES128、AES192、AES256 三种对称加密算法中选择其一。

-k (--encrypt-key=name)

指出加密或解密时要使用的密钥。

-f (--encrypt-key-file=name)

指出加密或解密时要使用的密钥文件。

-s (--encrypt-chunk-size=#)

设置加密时要使用的缓冲区大小，默认为 64 KB。

-v (--verbose)

输出加密或解密过程的详细处理内容。

4. 从服务器备份

MySQL 或 MariaDB 服务器通常均由主从服务器组成。在主 MySQL 服务器中实施备份会对其提供的服务产生不良影响，所以主要由从服务器进行备份操作。在从 MySQL 中使用 XtraBackup 进行备份时，必须知道备份时间点的主 MySQL 服务器的二进制日志文件与二进制日志的位置，才能构建新的从服务器或恢复主从 MariaDB 服务器。为此，XtraBackup 会将从服务器备份时间点的主 MariaDB 的二进制日志文件与位置记录到 xtrabackup_slave_info 文件。利用该信息可以搭建新的从服务器，或者在原从服务器故障时进行恢复备份，并在数据毫无损失的情形下进行连接复制。要使 XtraBackup 在从服务器备份时将主服务器的二进制日志文件与位置备份到 xtrabackup_slave_info 文件，必须使用 --slave-info 选项。

5. 并行备份

使用 xbstream 进行流备份时，如果使用 --parallel 选项，则同时可以复制多个数据文件。在 --parallel 选项中指定复制数据文件时要使用的线程数。由于并行（Parallel）备份功能以文件为单位进行工作，所以 MariaDB 服务器的数据文件聚集为一个文件或数据文件的大小差别相当大时，并行备份也可能不会有太大帮助。

除数据文件备份外，在 XtraBackup 中进行压缩时，使用 --compress-threads 选项也可以指定执行压缩的线程数。--parallel 选项与 --compress-threads 选项都可以用于设置线程数。

6. 调整备份速度

使用 XtraBackup 备份数据文件时，并不需要登录 MariaDB 服务器收集备份数据。它采用与 MariaDB 服务器无关的数据文件物理复制方式，所以对 MySQL 或 MariaDB 服务器的查询执行不会产生太大影响。但 XtraBackup 进行备份时，若耗尽了磁盘所有带宽，就会对 MariaDB 服务器的查询处理产生不良影响。

为了消除这一不良影响，XtraBackup 提供了调整备份速度的功能。使用 --throttle=1 选项时，XtraBackup 就会以秒为单位限制要复制的数据文件的大小。图 7-12 表示 XtraBackup 备份速度调整功能的工作原理。当然，实际情况可能并不像图中所示的那样每秒只读写磁盘 3 次。为了便于讲解，此处假设 1 秒钟读写 3 次磁盘。若将 throttle 设置为 0 或不设置任何值，XtraBackup 就会一直以最快的速度进行备份。若将 throttle 设置为 1，则进行 1 次以 1 MB 为单位的读写操作，在 1 秒内的其余时间处于等待状态。同样，若将 throttle 设置为 2，则进行 2 次 1 MB 的读数据文件与写备份文件操作，在 1 秒内的其余时间处于等待状态。

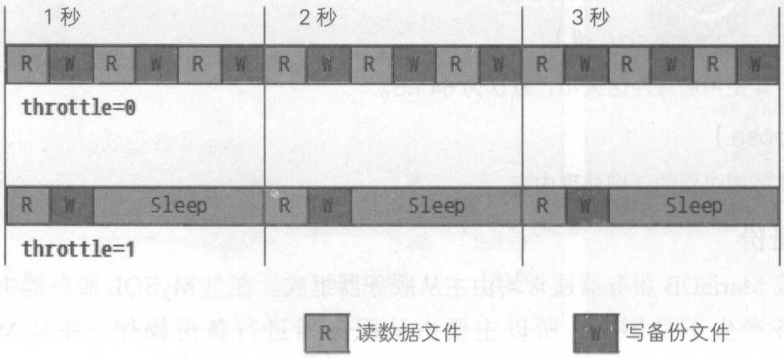


图 7-12 XtraBackup 调整备份速度的方式

请注意，throttle 选项的值并不是 TRUE 或 FALSE，它用于指定 1 秒内执行多少次 1 MB 单位的读写操作。此外，备份期间，由于 XtraBackup 要扫描并复制 InnoDB 或 XtraDB 的重做日志文件，所以判断磁盘 IO 可用量时要充分考虑这一点。

7. 恢复部分数据表

有时需要恢复备份数据的所有数据文件，而有时只需恢复部分数据表。假如备份文件的全部大小为 100 GB，而实际希望恢复的数据文件为 100 MB，则恢复全部备份并再次只复制相应数据表是相当不合理的。XtraBackup 中，可以只从备份文件中恢复所需部分数据表的数据文件。MySQL 5.6 或 MariaDB 10.0 开始添加了使用 ALTER TABEL ..[DISCARD TABLESPACE | IMPORT TABLESPACE] 命令导入 / 导出 InnoDB 数据表的表空间功能，所以可以使用非常有用的恢复部分数据表的功能。XtraBackup 的部分数据表恢复功能只能用于使用 InnoDB 或 XtraDB 存储引擎的数据表。

8. 改善 FLUSH TABLES WITH READ LOCK

从 MariaDB 执行备份时，为了在备份的最后时间点收集主 MySQL 服务器的二进制日志位置信息，需要加上 MySQL 服务器的全局锁（FLUSH TABLES WITH READ LOCK）。从服务器也经常用于长期执行统计或批查询处理。FLUSH TABLES WITH READ LOCK 锁命令会将其他连接中的新查询执行全部终止，并等待现有查询执行完毕。查询执行快速结束的系统中，使用 FLUSH TABLES WITH READ LOCK 命令不会有太大问题。但长时间执行的批处理或统计查询执行过程中，使用 FLUSH TABLES WITH READ LOCK 命令加锁时，只有长时间执行的查询全部完成才能获得全局锁。长时间执行的查询与 FLUSH TABLES WITH READ LOCK 锁命令经常相互引发问题，并且发生得相当频繁。因此，有时可能会因从服务器的复制延迟而需要进行复制重建，复制延迟也会使主从 MySQL 服务器的数据不一致。

以前版本的 XtraBackup 会无限期等待，直到获得全局锁。为了解决该问题，从 XtraBackup 2.1 开始可以使用以下两个选项，使 FLUSH TABLES WITH READ LOCK 命令获取全局锁。

等待

执行 FLUSH TABLES WITH READ LOCK 命令获得全局锁前，用户可以设置要等待的时间。

- --lock-wait-timeout

设置执行 FLUSH TABLES WITH READ LOCK 命令获得全局锁前要等待的时间，单位为秒。若超过该设置时间而仍存在长时间执行的查询（执行时间长于 --lock-wait-threshold 所设秒数的查询为长时间执行的查询），XtraBackup 就会终止执行。该设置的默认值为 0，此时，XtraBackup 将不等待而直接执行 FLUSH TABLES WITH READ LOCK 命令。

- --lock-wait-query-type

该选项可以设置为 all 与 update 两者之一。设置为 all 时，XtraBackup 执行 FLUSH TABLES WITH READ LOCK 命令前会一直等待，直到长时间执行（执行时间长于 --lock-wait-threshold 所设秒数的查询为长时间执行的查询）的所有类型（INSERT、UPDATE、DELETE、SELECT 等）的查询执行完毕。若设置为 update，则先等待除 SELECT 之外的 DML（UPDATE/ALTER/ REPLACE/INSERT）查询执行完毕，再执行 FLUSH TABLES WITH READ LOCK 命令。

强制终止

从执行 FLUSH TABLES WITH READ LOCK 命令开始到获得全局锁，在妨碍获得全局锁的查询中，也可以强制终止所有 SELECT 查询。

- --kill-long-queries-timeout

即使在强制终止模式下，对于执行时间短的查询，也要先短暂等待再执行 FLUSH TABLES WITH READ LOCK 命令。该选项用于设置 FLUSH TABLES WITH READ LOCK 命令执行完毕后短暂等待的秒数。

- --kill-long-query-type

该选项可以设置为 all 或 select。执行 FLUSH TABLES WITH READ LOCK 命令时，若因其他查询导致无法获得全局锁，可以使用该选项指定要强制终止所有查询还是只终止 SELECT 查询。

使用如下选项执行 XtraBackup 时，XtraBackup 不会为所有执行时间超过 40 秒的查询（--lock-wait-query-type）等待 600 秒（10 分钟）以上。执行 FLUSH TABLES WITH READ LOCK 命令后，将等待 30 秒以获得全局锁。如果 30 秒之内未能获得全局锁，XtraBackup 会将早于 FLUSH TABLES WITH READ LOCK 命令执行的所有查询（--kill-long-query-type）全部强制终止。

```
innobackupex --lock-wait-threshold=60 --lock-wait-query-type=all \
             --lock-wait-timeout=600 --kill-long-queries-timeout=30 \
             --kill-long-query-type=all /data/backups/
```

7.5.4 XtraBackup 的高级用法

前面讲解了 XtraBackup 的最基本的用法与功能。接下来学习使用 XtraBackup 进行增量备份或流式备份这类高级备份的方法。限于篇幅，此处省略备份或恢复过程的详细画面输出内容，只讲解进行备份或恢复的命令。

1. PIT 恢复

PIT 恢复只恢复特定时间点之前的数据。用户或管理员失误造成部分数据损坏或服务器故障导致数据文件损坏时，通常使用 PIT 恢复。进行 PIT 恢复时，首先必须有完全备份或增量备份，并且相应服务器的二进制日志文件必须为可读取状态。若因保存二进制日志文件的目录或文件系统损坏导致无法获取二进制日志文件，就无法使用 PIT 恢复。为了应对数据文件与二进制日志文件同时损坏的情形，需要将二进制日志文件实时备份到远程服务器，此时可以使用前面学习过的 MySQL 5.6 或 MariaDB 10.0 的 mysqlbinlog 实用工具。

下面简单介绍 PIT 恢复的使用方法。假设 2013 年 1 月 4 日 10 时整发生如下故障，存在完全备份与二进制日志文件。

```
故障发生时间：2013 年 1 月 4 日 10 时整
最后完全备份：/backup/2013-04-01-03-00-00
完全备份结束时二进制日志文件与位置 (mysql-bin.000003, 12345)
二进制日志文件：
  mysql-bin.000002 : 最后修改时间 (2013 年 1 月 4 日 02 时 20 分)
  mysql-bin.000003 : 最后修改时间 (2013 年 1 月 4 日 04 时 20 分)
  mysql-bin.000004 : 最后修改时间 (2013 年 1 月 4 日 07 时 30 分)
  mysql-bin.000005 : 最后修改时间 (2013 年 1 月 4 日 10 时整)
```

首先恢复 4 月 1 日 03:00 的完全备份。完全备份恢复后，在恢复的 MySQL 服务器中应用 mysql-bin.000003 的 12345 位置之前的二进制日志。因此，不再需要 mysql-bin.000003 之前的二进制日志文件。恢复的 MySQL 服务器中，只要使用 mysqlbinlog 实用工具将从 mysql-bin.000003 文件的 12345 位置开始的二进制文件形式的二进制日志转换为 SQL 文件并再次运行即可。

```
## 应用备份日志
```

```
shell> innobackupex --apply-log /backup/2013-04-01-03-00-00
## 使用备份恢复 MySQL 数据
shell> innobackupex --copy-back /backup/2013-04-01-03-00-00

## 启动 MySQL 服务器
shell> service mysql start

## 在启动的 MySQL 服务器中顺序应用二进制日志文件
shell> mysqlbinlog --start-position=12345 mysql-bin.000003 | mysql -u root -p
shell> mysqlbinlog mysql-bin.000004 | mysql -u root -p
shell> mysqlbinlog mysql-bin.000005 | mysql -u root -p
```

若用户或管理员误操作而执行了数据表删除命令，则要先去掉该命令，再应用二进制日志。此时，使用 mysqlbinlog 命令将删除命令执行时间生成的二进制日志转换为 SQL 文件。此外还要打开该文件，查找 DROP TABLE 命令，记录该事件之前与之后的事件的起始位置。若 DROP TABLE 命令记录在 mysql-bin.000005 文件中，DROP TABLE 命令之前的事件为 123456，之后的事件为 123489，则可以只删除 DROP 命令，再执行二进制日志，如下所示。

```
## 应用备份日志
shell> innobackupex --apply-log /backup/2013-04-01-03-00-00
## 使用备份恢复 MySQL 数据
shell> innobackupex --copy-back /backup/2013-04-01-03-00-00

## 启动 MySQL 服务器
shell> service mysql start

## 向启动的 MySQL 服务器顺序应用二进制日志文件
shell> mysqlbinlog --start-position=12345 mysql-bin.000003 | mysql -u root -p
shell> mysqlbinlog mysql-bin.000004 | mysql -u root -p

## 只执行到 mysql-bin.000005 二进制日志文件的 DROP TABLE 命令之前
shell> mysqlbinlog --stop-position=123456 mysql-bin.000005 | mysql -u root -p

## 从 mysql-bin.000005 二进制日志文件的 DROP TABLE 命令之后全部执行
shell> mysqlbinlog --start-position=123489 mysql-bin.000005 | mysql -u root -p
```

使用 mysqlbinlog 实用工具时，可以使用 --start-datetime 或 --stop-datetime 选项代替 --start-position 或 --stop-position 选项明确标注时间。但这样将时间明确指出后，可能丢失相同时间点执行的事件，也有可能再次执行，请各位注意。

2. 增量备份

顾名思义，增量备份只备份从之前执行完全备份的时间点开始更改的数据。增量备份与完

全备份一样，要全部读取实际要备份的数据文件。但读取的数据文件页面（块）中，只对序号大于完全备份时间点的日志序号的数据文件页面进行复制备份。因此，与完全备份相比，增量备份的执行速度不是非常快。由于增量备份只收集更改的部分，所以（根据服务器的工作量）备份文件可能非常小。

图 7-13 表示周日到周一数据文件的变化情况。增量备份中，由于只备份更改的数据，所以为了恢复备份，必须有以此为基准的完全备份，也需要仅收集从完全备份时间点开始变更的数据的增量备份。增量备份中，基准是 InnoDB/XtraDB 的日志序号（LSN）。前面基本备份与恢复中已经学过保存日志序号的位置，只有知道它才能进行增量备份。如图 7-13 所示，假设在数据变更状态中，每个周日凌晨 03:00 执行完全备份，其余每天的凌晨 03:00 执行增量备份，那么完全备份与增量备份的日志序号将如下所示。

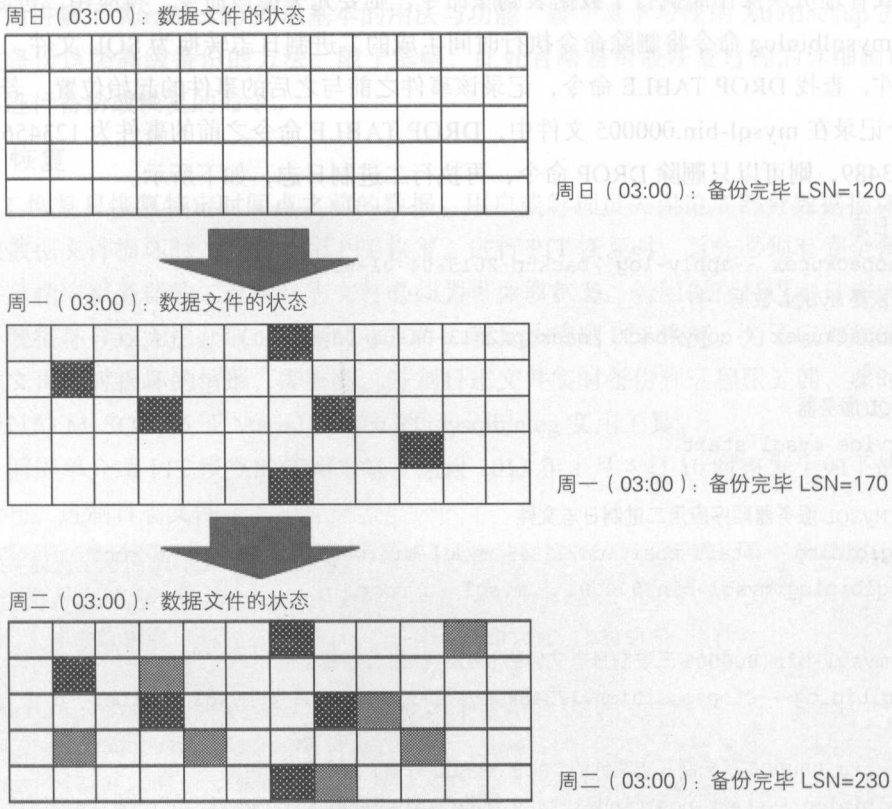


图 7-13 数据文件的变化

```
周日凌晨 03:00 完全备份：对日志序号在 0~120 之间的所有数据备份
周一凌晨 03:00 增量备份：对日志序号在 120~170 之间的所有数据备份
周二凌晨 03:00 增量备份：对日志序号在 170~230 之间的所有数据备份
```

增量备份中，备份起始日志序号总是大于 0；但完全备份中，起始日志序号总是为 0。下面

为周日、周一、周二要执行的备份命令。

周日凌晨备份

```
innobackupex \
  --host="127.0.0.1" \
  --user="backup_user" \
  --password='backup_user_password' \
  --port=3306 \
  --no-lock \                    ## 从服务器备份时删除 no-lock 选项
  --slave-info \                 ## 主服务器备份时删除 -slave-info 选项
  --defaults-file="/etc/my.cnf" \
  /backup/
```

周一凌晨备份

```
innobackupex \
  --host="127.0.0.1" \
  --user="backup_user" \
  --password='backup_user_password' \
  --port=3306 \
  --no-lock \                    ## 从服务器备份时删除 no-lock 选项
  --slave-info \                 ## 主服务器备份时删除 -slave-info 选项
  --defaults-file="/etc/my.cnf" \
  --incremental \
  --incremental-lsn=120 \        ## 周日完全备份完成时的日志序号（120）
  /backup/
```

周二凌晨备份

```
innobackupex \
  --host="127.0.0.1" \
  --user="backup_user" \
  --password='backup_user_password' \
  --port=3306 \
  --no-lock \                    ## 从服务器备份时删除 no-lock 选项
  --slave-info \                 ## 主服务器备份时删除 -slave-info 选项
  --defaults-file="/etc/my.cnf" \
  --incremental \
  --incremental-lsn=170 \        ## 周一完全备份完成时的日志序号（170）
  /backup/
```

以上就是依次执行完全备份与增量备份的方法，其实，使用该方法时每次必须查看以前的备份内容，以获取先前完全备份或增量备份结束时间点的日志序号。为此，XtraBackup 中可以

使用 incremental-basedir 选项代替 incremental-lsn 选项。在 incremental-basedir 选项中设置以前备份的路径，这比在以前备份中查找日志序号要更容易。每天的备份保存到如下目录时，增量备份命令如下所示。

- 周日备份：2014-01-05-03-00-00
- 周一备份：2014-01-06-03-00-00
- 周二备份：2014-01-07-03-00-00

```
## 完全备份命令相同

## 周一增量备份命令
innobackupex \
  --host="127.0.0.1" \
  ...
  --incremental \
  --incremental-basedir=/backup/2014-01-05-03-00-00 \        ## 周日完全备份路径
  /backup/

## 周二增量备份命令
innobackupex \
  --host="127.0.0.1" \
  ...
  --incremental \
  --incremental-basedir=/backup/2014-01-06-03-00-00 \        ## 周一增量备份路径
  /backup/
```

如上所示做好完全备份与增量备份后，下面学习恢复备份的方法。利用完全备份与增量备份进行恢复的过程类似于完全备份恢复，均使用 --apply-log 与 --copy-back 选项进行处理。但此处需要注意，最后一次增量备份之前的 --apply-log 过程中，必须使用 --redo-only 选项。

--apply-log 选项会再次运行（Roll forward）存档（备份过）的重做日志，维持备份数据文件的一致性。即使将存档的重做日志全部执行完，若仍有事务（未执行 COMMIT 或 ROLLBACK 的事务）没有结束，那么这些事务会全部回滚（ROLLBACK）。如图 7-13 所示，假设某个事务在完全备份结束前就已经开始，并且在完全备份结束后才提交。

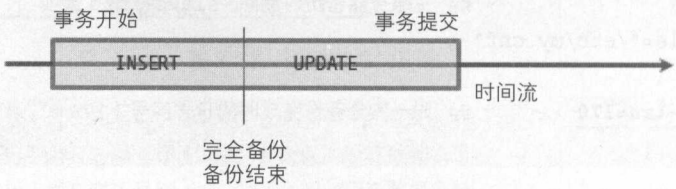


图 7-14 一个事务横跨增量备份结束点执行时

完全备份恢复过程中，执行 apply-log 时，连接备份结束时间点前后的事务全部回滚，如图

7-14 所示。希望对完全备份之后的增量备份执行 apply-log 时，会因无法查找到相应事务而导致 apply-log 操作失败，或者即使执行了 apply-log 操作，图 7-14 所示的事务数据也会全部消失。像这样，覆盖完全备份以后的增量部分的增量备份恢复过程中，必须维持原状，不要对尚未结束（提交或回滚）的事务进行回滚。使用 --redo-only 选项只会再次运行重做日志，而不会进行回滚。

下面简单介绍利用完全备份与增量备份进行恢复的过程。针对增量备份进行 apply-log 操作时，要在 --incremental-dir 选项中明确标注增量备份目录，备份目录必须明确指定为最初的完全备份目录。

```
## 周日凌晨的完全备份 apply-log
innobackupex --apply-log --redo-only  /backup/2014-01-05-03-00-00

## 周一凌晨的增量备份 apply-log
innobackupex --apply-log --redo-only  /backup/2014-01-05-03-00-00 \
                          --incremental-dir=/backup/2014-01-06-03-00-00

## 周二凌晨的增量备份 apply-log（由于周二增量备份是最后一个，所以此处删除 redo-only）
innobackupex --apply-log  /backup/2014-01-05-03-00-00 \
                        --incremental-dir=/backup/2014-01-07-03-00-00

## 在周日凌晨的完全备份中再次执行 apply-log
innobackupexp --apply-log /backup/2014-01-05-03-00-00
```

经过以上过程，apply-log 全部完成后，周一与周二的增量备份内容合并到周日的完全备份。如下所示，使用 copy-back 将完全备份的数据文件复制到 MySQL 或 MariaDB 服务器，调整数据文件的权限后，启动 MariaDB 服务器。

```
innobackupex --copy-back /backup/2014-01-05-03-00-00
```

3. 部分备份

使用 XtraBackup 也可以只备份部分数据表，而非整个数据库，这称为部分（Partial）备份。执行 XtraBackup 的 innobackupex 脚本时，可以使用 --include、--tables-file 以及 --databases 选项，在 --include 选项中可以使用正则表达式罗列要备份的目标数据库与数据表。使用 --tables-file 选项时，即使是很难使用正则表达式的情形，也可以逐个明确指出各数据库与数据表。--databases 选项进行以数据库为单位的备份。

使用正则表达式进行部分备份

使用正则表达式明确指出要备份的数据表时，“数据库名 . 数据表名”这种形式必须使用正则表达式表示。下列示例只备份 dbname 数据库中的 tablename 数据表。

```
shell> innobackupex \
  --host="127.0.0.1" \
```

```
  --user="backup_user" \
  --password='backup_user_password' \
  --port=3306 \
  --no-lock \                    ## 从服务器备份时删除 no-lock 选项
  --slave-info \                 ## 主服务器备份时删除 -slave-info 选项
  --defaults-file="/etc/my.cnf" \
  --include='^dbname[.]tablename'
  /backup/
```

以文件形式指出备份目标数据表

也可以将要备份的目标数据表以“数据库名 . 数据表名”形式在另外的文件中进行罗列，然后将文件路径在 innobackupex 脚本的 --tables-file 选项中明确指出。

```
shell> cat /tmp/backup_target.list
dbname1.tablename1
dbname2.tablename2

shell> innobackupex \
  --host="127.0.0.1" \
  --user="backup_user" \
  --password='backup_user_password' \
  --port=3306 \
  --no-lock \                    ## 从服务器备份时删除 no-lock 选项
  --slave-info \                 ## 主服务器备份时删除 -slave-info 选项
  --defaults-file="/etc/my.cnf" \
  --table-files=/tmp/backup_target.list
  /backup/
```

以数据库为单位进行备份

执行 innobackupex 脚本时，同时使用 --databases 选项也能以数据库为单位进行备份。当然，此时也可以只备份特定数据库中的部分数据表。下列示例中，只备份 daname1 数据库中的 tablename1 数据表，同时备份 dbname2 数据库中的所有数据表。

```
shell> innobackupex \
  --host="127.0.0.1" \
  --user="backup_user" \
  --password='backup_user_password' \
  --port=3306 \
  --no-lock \                    ## 从服务器备份时删除 no-lock 选项
  --slave-info \                 ## 主服务器备份时删除 -slave-info 选项
  --defaults-file="/etc/my.cnf" \
  --databases='dbname1.tablename1  dbname2'
  /backup/
```

像这样创建部分备份后，进行恢复时，与完全备份或增量备份不同，必须使用以数据表为单位的恢复方式。逐表恢复方式中，执行 innobackupex 脚本时必须一起使用 --apply-log 与 --export 选项。

```
shell> innobackupex --apply-log --export /data/2013-04-01-03-00-00
```

使用 --export 选项时，执行 apply-log 的过程中会输出如下信息，并以备份的数据表为单位生成 .exp 与 .ibd 及 .cfg 文件。利用这些文件，以 MySQL 5.6 服务器的表空间交换方式，将备份的数据表应用到 MariaDB 或 MySQL 服务器。

```
..
xtrabackup: export option is specified.
xtrabackup: export metadata of table 'dbname1/tablename1' to file
'./dbname1/tablename1.exp' (1 indexes)
..
```

如下所示，利用准备好的 tablename1.cfg 与 tablename1.ibd 文件，以各数据表为单位采用交换表空间的方式进行恢复。

```
shell> ALTER TABLE dbname1.tablename1 DISCARD TABLESPACE;
shell> ALTER TABLE dbname1.tablename1 IMPORT TABLESPACE;
```

4. 精简备份

运行 XtraBackup 这类物理备份工具进行备份时，所有数据库的磁盘文件都会得到原样复制。与此不同，类似于 mysqldump 的逻辑备份工具实际只备份数据表的数据。因此，与 XtraBackup 这类物理备份工具相比，mysqldump 这类逻辑备份工具产生的备份会更小。特别是数据表建有大量索引时，使用逻辑备份工具创建的备份文件明显更小。

从 XtraBackup 2.1 开始，精简（Compact）备份成为可能。进行物理备份的同时，可以忽略所有索引页而只备份数据表的数据区域。要进行精简备份，则执行 innobackupex 脚本的同时必须使用 --export 选项。

```
shell> innobackupex \
  --host="127.0.0.1" \
  --user="backup_user" \
  --password='backup_user_password' \
  --port=3306 \
  --no-lock \                    ## 从服务器备份时删除 no-lock 选项
  --slave-info \                 ## 主服务器备份时删除 -slave-info 选项
  --defaults-file="/etc/my.cnf" \
  --compact \
  /backup/
```

使用 --compact 选项进行备份时，由于索引区域并不包含于实际的备份数据文件，所以备份很小，不需要大量磁盘空间。但恢复备份时，必须恢复未备份的全部索引区域，进行 --apply-log 操作的过程中再次生成索引。因此，精简备份虽然可以减小备份大小，但恢复过程相当长。使用 --compact 选项进行备份时，备份目录下 xtrabackup_checkpoints 文件的 compact 选项设置为 1。恢复时，通过该值判断是否新建索引。

```
shell> cat xtrabackup_checkpoints
backup_type = full-backuped
from_lsn = 0
to_lsn = 2888984349
last_lsn = 2888984349
compact = 1
```

5. 流备份

流备份可以用于完全备份或增量备份等不同类型的备份。流备份其实并不是一种备份类型，它将备份数据传送到本地磁盘或远程服务器。备份数据量很大导致磁盘存储空间不足或要进行加密时，一定需要使用流备份。

下面学习使用流备份将备份直接传送到远程服务器并存储（非本地磁盘保存）的方法。前面已经学过流备份的工作原理，此处只讲解执行流备份的命令。

```
shell> innobackupex \
  --host="127.0.0.1" \
  --user="backup_user" \
  --password="backup_password" \
  --port=3306 \
  --defaults-file="/etc/my.cnf" \
  --stream=xbstream \
  /backup/ \
  | ssh -o StrictHostKeyChecking=no   remote_user@remote_host \
          "cat - > /backup/20130401.xbs"
```

上述命令中的 --stream=xbstream 选项将备份结果进行流输出。--stream 选项要给出流归档的实用工具名。可以与 XtraBackup 一起使用的归档实用工具只有 xbstream（虽然也有变形后的 tar，但现在几乎不用）。由于 xbstream 实用工具包含在 XtraBackup 包中一同发布，故不需另行安装。xbstream 的归档备份流通过 ssh 传送到 remote_host，此时，remote_user 不是 MySQL 服务器的用户，而是 Unix 操作系统的用户名，请各位不要混淆。请不要将上述命令中的“-”（标准输入标记）与“>”（重定向标记）符号搞混，也不要漏掉命令中的空格。顺序不对时，可能损坏备份。remote_host 未注册为已知主机（Known_hosts）时，SSH 的“-o StrictHostKeyChecking=no”选项删除询问是否保存键文件的对话模式。

为了使用 xbstream 实用工具从归档备份开始恢复，首先必须将归档文件还原为普通文件。

此时也使用 xbstream 实用工具。下列命令将以 20130401.xbs 文件形式存在的归档文件还原到 /backup/extracted 目录。以下两个命令功能相同，可以选择其一。使用 xbstream 解档后，可以查看 XtraBackup 备份的目录结构。

```
shell> cat 20130401.xbs | xbstream -x -C /backup/extracted/
shell> xbstream -x -C /backup/extracted < 20130401.xbs
```

像这样将备份解档后，再执行 innobackupex --apply-log 与 innobackupex --copy-back 命令进行恢复。

6. 加密备份

加密（Encrypt）备份时，先将备份结果加密保存，再次恢复时进行解密，并执行 apply-log 与 --copy-back 操作。当然，可以借助流备份，使用第三方解决方案对备份进行加密，但 XtraBackup 本身就带有备份加密功能。XtraBackup 2.1.4 支持 AES128、AES192、AES256 三种加密算法。为了使用加密备份，首先要根据所用算法生成相应位数的密钥。而要生成密钥，需要使用 openssl 命令。

```
shell> openssl enc -aes-256-cbc -pass pass:yourpassword -P -md sha1
```

openssl 命令中，enc 选项用于生成对称加密密钥，-aes-256-cbc 选项为 AES 256 加密算法的 CBC 模式生成 256 位密钥。若 XtraBackup 加密时不采用 AES 256 而采用 AES 128 加密算法，则只需使用 -aes-128-cbc 选项。-P 选项将生成的密钥内容输出到画面，-md 选项是消息摘要（Message Digest）算法，示例中明确指定使用 sha1。openssl 已超出本书讨论范围，更详细的内容请各位参考相关网站。执行上述命令，画面输出如下内容。输出的内容中，将 iv 项的值用作 XtraBackup 的密钥。

```
salt=A842CC4A0D4A8AA5
key=919EA523F3206FCE66246B983227B45D5F257E3C980CDE1907A4DF7A76FD07BF
iv =6004EF32FE1ACFD04C1FB955FAC136D5
```

注意 openssl enc 命令示例中，生成密钥时，通过 -pass 选项输入密码“pass:yourpassword”。此时，实际密码部分为“yourpassword”，其前面的“pass:”是必须使用的关键字。若密码为“abc”，则要使用的选项为“-pass pass:abc”。

下面使用生成的密钥执行加密备份。使用 XtraBackup 进行加密备份时，既可以将密钥用作命令行参数，也可以将密钥保存到单独的文件，然后将文件路径设置为命令参数。

```
## 将密钥直接用作命令行参数
shell> innobackupex \
  --host="127.0.0.1" \
  --user="backup_user" \
  --password="backup_password" \
  --port=3306 \
```

```
  --defaults-file="/etc/my.cnf" \
  --encrypt=AES256 \
  --encrypt-key="6004EF32FE1ACFD04C1FB955FAC136D5" \
  /backup/

## 将密钥保存为单独文件
## echo 命令的 -n 选项的句末不要包含换行字符
shell> echo -n "6004EF32FE1ACFD04C1FB955FAC136D5" > /backup/encrypt_key
shell> innobackupex \
  --host="127.0.0.1" \
  --user="backup_user" \
  --password="backup_password" \
  --port=3306 \
  --defaults-file="/etc/my.cnf" \
  --encrypt=AES256 \
  --encrypt-key-file=/backup/encrypt_key \
  /backup/
```

加密后生成的所有备份数据文件的扩展名均为 .xbcrypt，解密数据文件时，只要执行 innobackupex --decrypt 命令即可。执行 innobackupex --decrypt 命令时，同时要使用 --parallel 选项。像 --parallel=4 一样，将该选项值设置为 4 后，将会有 4 个线程同时进行解密处理。

```
shell> innobackupex --decrypt=AES256 \
         --encrypt-key="6004EF32FE1ACFD04C1FB955FAC136D5" /backup/2013-04-01_03-00-00/

shell> innobackupex --decrypt=AES256 \
         --encrypt-key-file=/backup/encrypt_key  /backup/2013-04-01_03-00-00/
```

对加密的备份解密后，删除备份目录中的所有 .xbcrypt 文件，并创建加密前的 .ibd 文件。接下来就像普通完全备份恢复一样，使用 apply-log 与 copy-back 命令进行恢复即可。

第8章

复制

8.1 全局事务ID MariaDB MySQL

8.2 多源复制 MariaDB

8.3 多线程复制 MariaDB MySQL

8.4 Crashsafe Slave MariaDB MySQL

8.5 改善基于ROW的复制功能 MariaDB MySQL

8.6 延迟复制 MySQL

8.7 MariaDB与MySQL服务器间的复制

8.8 其他功能改善 MariaDB MySQL

复制是MariaDB中实现高可用性及负载平衡的必需功能。人们长期使用基于STATEMENT的复制，其具有良好的稳定性。为解决STATEMENT问题而引入的基于ROW的复制等许多功能都得到了改善，但用户对复制功能的许多要求还是没能得到满足。为了满足用户的这些要求，MariaDB 10.0与MySQL 5.6对复制功能有大幅改善，也添加了许多新功能。

许多用户长期要求的GTID（Global transaction ID，全局事务ID）与多源复制功能都已添加到MariaDB 10.0。多源复制只有MariaDB 10.0支持，MySQL的多源复制功能目前正在开发，预计5.7版本中会加入。当然，此外，有关实现稳定复制与基于ROW复制容量的问题在一定程度上都得到了解决。MariaDB 10.0与MySQL 5.6的复制中需要注意Multi-threaded slave。当然，Multi-threaded slave尚未达到令人满意的程度，但Monty Program AB与Oracle正在积极改善该功能，令人充满期待。

本章将讲解MariaDB 10.0与MySQL 5.6中复制功能相关的新技术，也将学习有关MariaDB与MySQL之间复制构成的兼容内容。此外，还要学习有关混用二进制日志的内容。

8.1 全局事务ID MariaDB MySQL

MariaDB（MySQL）5.5以前的版本中，也像MariaDB 10.0或MySQL 5.6一样，为了复制而将主MySQL中执行的事务信息记录到二进制日志。但MariaDB 5.5中并不会为各事务或DML（INSERT、UPDATE、DELETE等）语句赋予单独的ID。因此，MariaDB 5.5中，为了获取主服务器的二进制日志事件，会使用传递二进制日志文件名称与偏移位置（Offset）这一原始方法。这种只通过文件名与文件内位置识别事件的原始方法只在相关本地服务器的文件中有意义。其实，相同的二进制事件在从MariaDB中并不一定保存在相同文件名的同一位置。简言之，原有二进制日志文件名与位置信息和投入复制的服务器之间完全不兼容。

利用参与复制的主从MariaDB服务器之间不兼容的信息构建复制系统时，经常会导致复制的拓扑无法变更。参与复制的服务器中，部分服务器发生故障时，需要变更复制的拓扑结构，若此时拓扑结构很难更改，就意味着很难通过复制进行故障恢复（Fail-over）。因此，MHA或MMM这类MariaDB HA（High Availability）解决方案内部会计算复杂二进制日志文件的位置，处理时偶尔也会彻底抛弃。

事实上，全局事务ID并不是某天引入的新技术。引入全局事务ID的呼声由来已久，而只有MariaDB 10.0与MySQL 5.6引入了该技术。下面学习全局事务ID的定义及用途。全局事务ID的基本概念在MariaDB 10.0与MySQL 5.6中都是一样的，但实际的内部实现不同，管理命令或方式也略有不同。后面还会介绍MySQL 5.6与MariaDB 10.0的全局事务ID的使用方法。

8.1.1 全局事务ID的定义

MariaDB 5.5中的二进制日志文件名或偏移位置指的是保存相关事件的物理位置。而全局事

务 ID 是一个逻辑上的概念，与物理文件名或位置完全无关。MariaDB 10.0 与 MySQL 5.6 中的所有全局事务 ID 都分为 2~3 个区域，主要由服务器的节点 ID 与事务 ID 组成。对于节点 ID，MariaDB 10.0 与 MySQL 5.6 的组成方式有差异，但事务 ID 都是一样的，全是 8 字节常数简单增加的形式。由于全局事务 ID 由节点 ID 与事务 ID 组合而成，所以主从服务器之间不会有重复，在一个集群（参与复制的节点组）中将有唯一 ID。

图 8-1 表示全局事务 ID 与原有的二进制日志事件组合使用的过程，虽然以 MariaDB 10.0 的全局事务 ID 为基准进行表达，但重要概念仍然与 MySQL 5.6 一样。

首先，主 MariaDB 中每当发生事务、DML（INSERT、UPDATE、DELETE 等）、DDL 时，事务 ID 就会加 1，同时生成全局事务 ID，并与二进制日志事件映射。这样记录下来的二进制日志内容会采用与 MariaDB 5.5 相同的方式传递给从服务器，从服务器接收到的事件会再次记录到延迟日志文件或从服务器的二进制日志文件。此时，延迟日志或二进制日志的文件名及事件的存储位置（文件内的 Offset）与主服务器可能不同，但全局事务 ID 的值保持不变。



GTID	二进制日志文件		事件
	文件名	位置	
0-1-1	binary-log.000002	391	INSERT INTO test VALUES(11);
0-1-2	binary-log.000002	589	INSERT INTO test VALUES(12);
0-1-3	binary-log.000002	787	INSERT INTO test VALUES(13);
0-1-4	binary-log.000002	947	INSERT INTO test VALUES(14);
	binary-log.000002	1145	INSERT INTO test VALUES(15);
0-1-5	binary-log.000002	1381	INSERT INTO test VALUES(16);



GTID	二进制日志文件		事件
	文件名	位置	
0-1-1	binary-log.000021	123591	INSERT INTO test VALUES(11);
0-1-2	binary-log.000021	123789	INSERT INTO test VALUES(12);
0-1-3	binary-log.000021	123987	INSERT INTO test VALUES(13);
0-1-4	binary-log.000021	124147	INSERT INTO test VALUES(14);
	binary-log.000021	124345	INSERT INTO test VALUES(15);
0-1-5	binary-log.000021	124581	INSERT INTO test VALUES(16);

图 8-1 全局事务 ID 概要

其实，全局事务 ID 的概念非常简单，它只是在原来 MariaDB 5.5 的二进制日志文件中多保存了一个全局事务 ID。但是，为了在主从服务器间保持全局事务 ID 的一致性，MariaDB 与 MySQL 服务器的代码需要进行许多改变。这一简单功能使复制相关命令的使用方法全部发生改变。

图 8-1 中还需要注意，MariaDB 5.5 的二进制日志文件名与位置信息并没有与某个固定的全局事务 ID 进行 1:1 映射。图 8-1 对于 GTID 为 0-1-4 的记录而言，二进制日志文件 binary-

log.000002 的 947 与 1145 号位置都连接到同一 GTID（0-1-4）。虽然每个事件（一般而言是一个 SQL 语句）都连接到一个二进制日志文件的偏移位置，但全局事务 ID 并不以 SQL 语句为单位进行分配，而是以事务为单位进行分配。但 MariaDB 10.0 或 MySQL 5.6 中，使用不支持事务的存储引擎时，GTID 会以 SQL 语句为单位进行分配。

MariaDB 10.0 与 MySQL 5.6 中的全局事务 ID 并不采用组合方式，在概念上也有些不同。MariaDB 10.0 与 MySQL 5.6 中的所有全局事务 ID 均由服务器唯一识别 ID 与服务器中单增的事务 ID 组成[①]。各服务器中单增形态的事务 ID 都从 1 开始，每个事务、DML 或 DDL 会增加 1。MySQL 5.6 中不允许单增 1 的事务 ID 在中间漏掉值。也就是说，事务 ID 为 1~10，漏掉 10 后将无法从 11 开始。但 MariaDB 10.0 的全局事务 ID 中，事务 ID 没有这种严格的限制，即可以生成遗漏的事务号码。这一特点使 MariaDB 10.0 与 MySQL 5.6 中使用全局事务 ID 进行复制时忽略并跳过特定二进制日志事件的方法变得不同。

此外，全局事务 ID 与 MySQL 5.6 及 MariaDB 10.0 的多线程复制功能相结合。MySQL 5.6 的二进制日志事件的运行可以与顺序无关，但 MariaDB 10.0 中，从一个主服务器接收的二进制日志一定要根据顺序运行。更准确地说，MariaDB 10.0 中若域标识符相同，二进制日志就要按顺序运行。因此，MySQL 5.6 与 MariaDB 10.0 的复制管理方式出现差异，更详细的内容将在多线程从服务器中进行讲解。

8.1.2 全局事务 ID 的必要性

凡是对复制结构或故障恢复策略有过思考的用户，可能都会清楚地知道 MariaDB 5.5 中有关复制的问题。下面简单介绍常用的复制拓扑结构。图 8-2 所示的复制拓扑结构中，一个主服务器与两个从服务器连接在一起。为了减轻从服务器读取负荷以及用于统计或批处理时，大量采用这种拓扑结构。图 8-2 中，当前主服务器的二进制日志位置是 binary-log.000002:320，从服务器 B 保持完全同步，所以处于完全执行结束到 binary-log.000002:320 二进制日志事件的状态。B 服务器用于分担 SELECT 查询，C 从服务器用于批处理或统计。但从服务器 C 略微有些延迟，复制只同步到主服务器的 binary-log.000002:120。

假设此时主服务器 A 出现故障且异常终止，那么 B 与 C 从服务器之中有一个会被提升为主服务器，并且原来连接到 A 服务器的客户端连接会转移到新的主服务器。此时当然应该将处于完全同步状态的 B 从服务器提升为主服务器。接下来，如图 8-3 所示，复制全部断开，客户端的流量流入 B 服务器。然而由于 C 服务器仍然处于异步状态，所以无法提供 SELECT 服务。

服务器 B 提升为新的主服务器后，客户端的查询请求就开始进入 B 服务器。由于 B 服务器先前用于分担 SELECT 查询负荷，所以 SELECT 与原有主服务器作用重叠，服务器 B 开始进入超负荷状态。此时，只要将 B 服务器的 SELECT 查询转移到 C 服务器即可。但令人遗憾的是，由于 A 服务器终止时，服务器 C 处于异步状态，所以无法将复制同步到最后的时间点。

① 详细内容将在各全局事务 ID 说明中进行讲解。

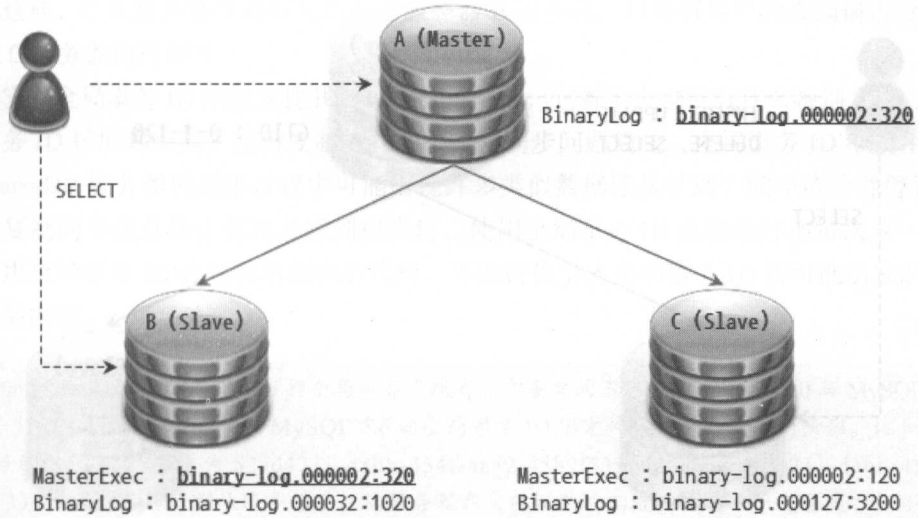


图 8-2 故障发生前的复制拓扑

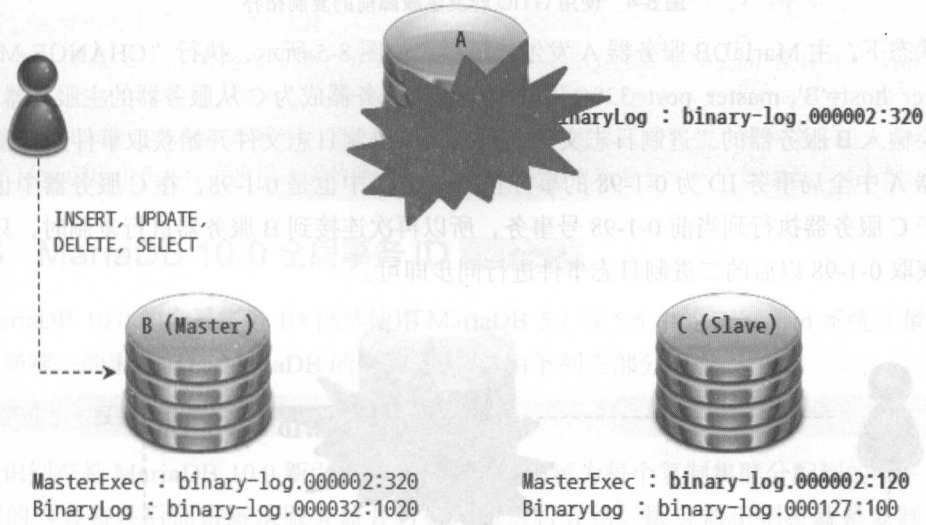


图 8-3 故障发生后服务器 B 提升为主服务器

当然，查找 C 服务器执行的最后事件（120 号位置的）的查询语句，在 B 服务器的二进制日志文件中查找相同语句的准确位置，然后从该位置开始进行连接复制。然而这个问题并不像说的那样简单，而且几乎不可能将该过程自动化。像这样，在简单的复制系统中进行恢复也并非易事。图 8-2 中，若从服务器再次连接到 B 或 C 服务器而用于复制，会怎么样呢？事实上，这种情形下，主服务器异常终止后，可能只能重新搭建新的复制系统。

下面考虑使用全局事务 ID 进行复制的情形。如图 8-4 所示，首先使用全局事务 ID 构筑复制系统，主 MariaDB 服务器目前的全局事务 ID 为 0-1-120，从服务器 B 完全同步到 0-1-120 的状态，从服务器 C 只同步到 0-1-98 号全局事务 ID。

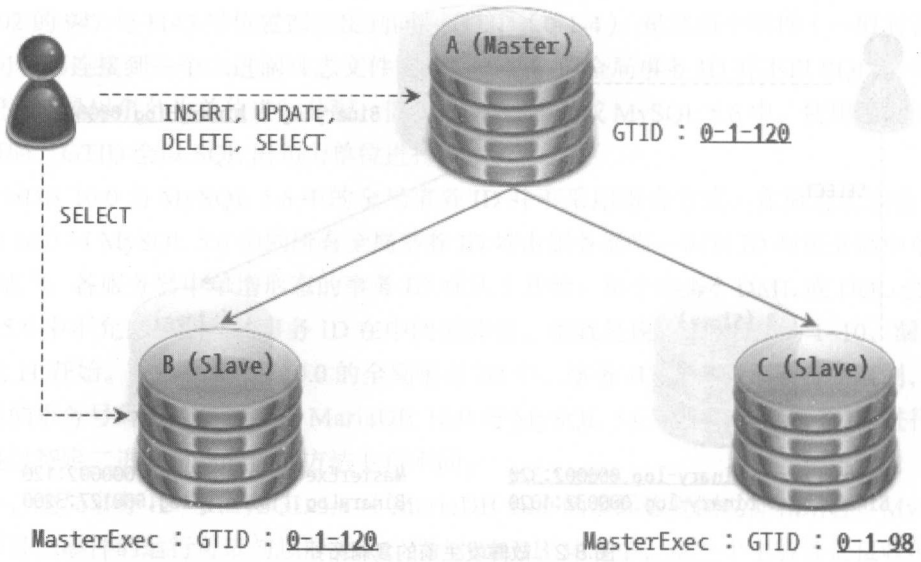


图 8-4 使用 GTID 时发生故障前的复制拓扑

该状态下，主 MariaDB 服务器 A 发生故障后，如图 8-5 所示，执行“CHANGE MASTER TO master_host='B', master_port=3306;”命令，使 B 服务器成为 C 从服务器的主服务器。此时，没有必要输入 B 服务器的二进制日志文件名以及从二进制日志文件开始获取事件的位置。因为在服务器 A 中全局事务 ID 为 0-1-98 的事务在 B 服务器中也是 0-1-98，在 C 服务器中也是 0-1-98。由于 C 服务器执行到当前 0-1-98 号事务，所以再次连接到 B 服务器进行复制时，只要从 B 服务器获取 0-1-98 以后的二进制日志事件进行同步即可。

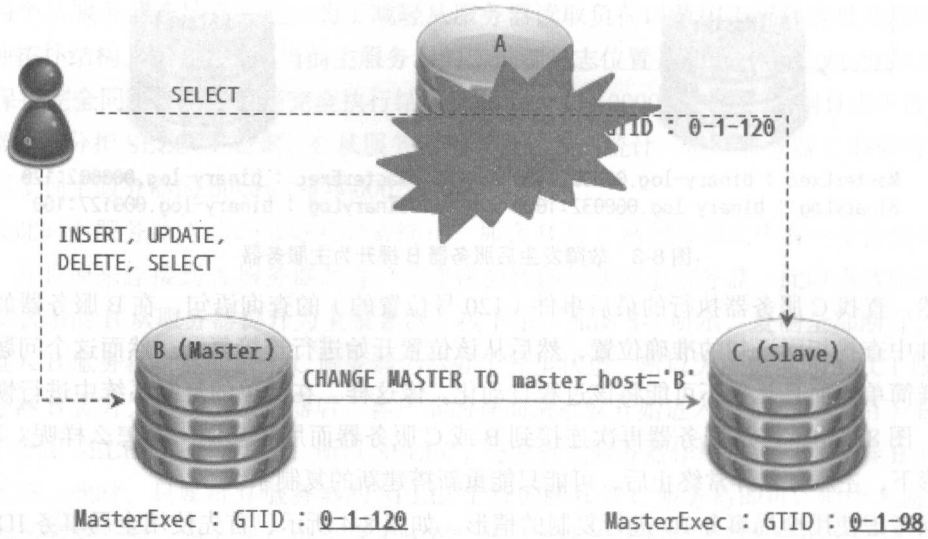


图 8-5 使用 GTID 时发生故障后 B 服务器提升为主服务器

像这样，C 从服务器准备好与新的主服务器 B 同步后，只要将用户的查询请求分派给 B 服务器与 C 服务器执行即可。

其实，全局事务 ID 的概念比我们想象得更简单，它看上去也没有什么特别之处。但是，通过将事务 ID 扩展到全局，使得复制拓扑变更时的同步问题简化。全局事务 ID 不仅用于应对故障，MariaDB 服务器的运维过程中可能出现许多类似数据库从扩展、缩小或合并等紧要问题，此时，复制同步化总是让管理者感到很头疼，使用全局事务 ID 也能很好地解决这一问题。当然，使用全局事务 ID 必须付出相应的代价。下面讨论引入全局事务 ID 后可能引发的问题，并学习相关内容。

注意 MariaDB 10.0 与 MySQL 5.6 的全局事务实现有非常重要的不同。MariaDB 10.0 与 MySQL 5.6 的全局事务 ID 都是顺次增 1 的，但 MySQL 5.6 的全局事务 ID 不允许中间出现遗漏的情形。比如，主服务器传递的二进制日志流为 52ad4731-d3bb-454f-ae82-4889f5338954:1~ 52ad4731-d3bb-454f-ae82-4889f5338954:100，传送到从服务器后，从服务器在自身记录的二进制日志中一定也是 52ad4731-d3bb-454f-ae82-4889f5338954:1~52ad4731-d3bb-454f-ae82-4889f5338954:100，且中间不允许遗漏。但就 MariaDB 的全局事务 ID 而言，即使主服务器中发生了 0-1-1~0-1-100 的事件，从服务器的二进制日志记录时，也可以将中间的 0-1-30 漏掉。

这个差异看上去并不怎么重要，但这种不同会使 MySQL 5.6 与 MariaDB 10.0 的 Multi-threaded slave 工作方式变得不一样。此外，由于 MySQL 5.6 的全局事务 ID 中不允许中间发生遗漏，所以从服务器不能跳过一个二进制日志事件。因此，MySQL 5.6 中只有采用特意插入线程事务的方式才能跳过。但由于 MariaDB 10.0 允许有中间遗漏的事务编号，所以复制中断后可以从任何位置开始复制。

8.1.3 MariaDB 10.0 全局事务 ID MariaDB

MariaDB 10.0 的全局事务 ID 仍然使用 MariaDB 5.1 或 5.5 中的 server_id 系统变量，生成集群中全局唯一的事务 ID。MariaDB 的全局事务 ID 由下列三部分组成。

```
" 域识别符 "-" 服务器识别符 "-" 事务识别符 "
```

域识别符是 MariaDB 10.0 新引入的概念，将其理解为每个复制集群分配到的唯一 ID 即可，更详细的内容将在后面讲解从服务器并行处理时进行介绍。服务器识别符就是复制中使用的 server_id 系统变量。最后事务识别符是依次增 1 的 8 字节常数值。其实，一个复制集群中，各 MariaDB 服务器必须使用唯一的服务器识别符，所以服务器识别符与事务识别符的组合也可以用作全局事务 ID。但 MariaDB 10.0 中引入了多源复制（Multi source replication）功能，一个从服务器可以同时对多个主服务器的数据进行备份。因此，只使用服务器识别符与事务识别符就不能再将各事务在全局范围内区分开来，为此引入“域识别符”概念。

定义固有的 server_id 系统变量后，默认激活 MariaDB 10.0 的全局事务 ID 功能。也就是说，从 MariaDB 5.5 升级到 MariaDB 10.0 后，不需要做特别设置，默认可使用全局事务 ID。MariaDB 10.0 中，若未在 MariaDB 的设置文件中另行设置 server_id，则 server_id 系统变量会初始化为默认值 1，但其实并未激活全局事务 ID 功能。因此，若未在 MariaDB 设置文件中明确设

置 server_id，则在服务器启动状态下，从服务器中执行 CHANGE MASTER 命令就会出现如下错误。

```
Errno: 1236
Error: Got fatal error 1236 from master when reading data from binary log:
    'Misconfigured master - server id was not set'
```

此外，MariaDB 10.0 中，server_id 系统变量成为动态变量，所以即使不重启服务器也可以随时调整 server_id 变量的值。

1. 使用 GTID 搭建复制系统

其实，在 MariaDB 10.0 中搭建使用全局事务 ID 的复制系统与在 MariaDB 5.5 服务器中没有太大区别。

将基于二进制日志文件与位置的复制变更为 GTID

从 MariaDB 5.5 升级为 MariaDB 10.0 后，服务器在不需要做任何设置的情形下即可使用全局事务 ID。从服务器与主服务器的复制虽然基于二进制日志文件与位置进行，但也会从主服务器一起接收 GTID 进行追踪。此情形下，在从服务器中执行一次下列命令即可。

```
MariaDB> STOP SLAVE;

MariaDB> CHANGE MASTER TO master_host='',master_port=3306,
                master_user='',master_password='', master_use_gtid=current_pos;

MariaDB> START SLAVE;
```

使新从服务器使用 GTID

向拥有数据的 MariaDB 10.0 添加从服务器的步骤如下。下面过程中，备份并恢复原有 MariaDB 服务器数据的过程请参考 7.5 节。

(1) 备份原 MariaDB 服务器的数据；
(2) 在新从 MariaDB 服务器中恢复备份数据；
(3) 查看备份时间点二进制日志文件名与位置；
(4) 利用步骤 3 中的二进制日志文件名与位置查看主服务器的 GTID；
(5) 将新从服务器的 GTID 值修改为步骤 4 中的 GTID；
(6) 在新从服务器中执行 CHANGE MASTER 命令。

上述利用全局事务 ID 搭建备份系统的过程中，与原有 MariaDB 5.5 的不同之处是步骤 3 与 4。使用 XtraBackup 2.1.4 与 MariaDB 10.0.8 中包含的 mysqldump 实用工具进行备份时，不会备份 MariaDB 服务器的全局事务 ID。因此，必须使用原有的二进制日志文件名与位置求得全局事务 ID，此时使用 BINLOG_GTID_POS() 函数。若主服务器中备份时间点的二进制日志位置为 binary-log.000002:1145，则调用执行 BINLOG_GTID_POS() 函数即可访问全局事务 ID。

```
MariaDB> SELECT BINLOG_GTID_POS("binary-log.000002", 1145);
+-----------------------------------------+
| BINLOG_GTID_POS("binary-log.000002", 1145) |
+-----------------------------------------+
| 0-1-4                                   |
+-----------------------------------------+
```

从运行结果可知，二进制日志文件 binary-log.000002 中，位于 1145 偏移位置的事件全局事务 ID 是 0-1-4。即使获知了全局事务 ID，也无法使用它直接执行 CHANGE MASTER 命令。在 MariaDB 10.0 中使用全局事务 ID 进行连接复制时，采用下列三种方法之一。

```
## MariaDB 5.5 与之前版本使用的传统方法
CHANGE MASTER TO master_host='',master_port=3306,
                 master_user='',master_password='', master_log_file='',
master_log_pos='';

## MariaDB 10.0 开始可用的方法
CHANGE MASTER TO master_host='',master_port=3306,
                 master_user='',master_password='', master_use_gtid=current_pos;

CHANGE MASTER TO master_host='',master_port=3306,
                 master_user='',master_password='', master_use_gtid=slave_pos;
```

上述示例中，第一个示例明确指出二进制日志文件名与二进制日志文件位置，它是我们非常熟悉并且一直在使用的方法。虽然该方法实际并不使用全局事务 ID 进行复制，但在复制处于连接的状态下，随时可以转换为使用全局事务进行复制。第二个与第三个示例从 MariaDB 10.0 开始才可以使用，CHANGE MASTER 命令的 master_use_gtid=current_pos 与 master_use_gtid=slave_pos 选项都采用原字符串。第二个与第三个方法借助全局事务 ID 进行复制。在 SHOW SLAVE STATUS 命令的 Using_Gtid 列中可以看到当前复制正在使用的全局事务 ID。若 Using_Gtid 列值为 No，则表示不使用全局事务 ID 进行复制。

第二个与第三个示例的 master_use_gtid 选项使用的 slave_pos 与 current_pos 含义如下。

master_use_gtid=slave_pos

使用全局事务 ID 初次构建复制时，如果得知复制要开始的全局事务 ID，则可使用该选项。执行 CHANGE MASTER 命令时，一起使用 master_use_gtid=slave_pos 选项，MariaDB 服务器使用 gtid_slave_pos 系统变量设置的 GTID 值连接主服务器进行复制。由于 gtid_slave_pos 系统变量是动态变量，必要时可以使用 SET 命令进行修改。

master_use_gtid=current_pos

使用全局事务 ID 进行复制的环境下，从未担当过从服务器的 MariaDB 服务器中，gtid_slave_pos 系统变量没有设置任何值。比如，一个复制系统中，A 是主服务器，B 是从服务器，若服务器 A 只用作主服务器，则其 gtid_slave_pos 系统变量为空。但此时若想切换主从服务器，使服务器 B 提升为主

服务器，服务器 A 成为从服务器，就不能在 CHANGE MASTER 命令中使用 master_use_gtid=slave_pos 选项。

此情形下，只要使用 CHANGE MASTER 与 master_use_gtid=current_pos 选项即可。使用 master_use_gtid 选项，MariaDB 服务器可以从 gtid_current_pos 系统变量中保存的 GTID 开始复制。MariaDB 服务器中，自身执行且记录在二进制日志中的事务的 GTID 值总是存储于 gtid_current_pos 系统变量。若服务器 A 是主服务器，则最后执行的 GTID 值保存到 gtid_current_pos 系统变量。因此，在服务器 A 中执行 CHANGE MASTER .. master_use_gtid= current_pos 命令时，服务器 A 会从服务器 B 获取迄今为止执行的事务以后的事务。

无论使用上述三种方法中的哪一种，MariaDB 从服务器每次从主 MariaDB 服务器获得二进制日志事件时，都会一起接收全局事务 ID。这样，从服务器会一直追踪接收的全局事务 ID，并将相关信息存储到 gtid_slave_pos 与 gtid_current_pos 系统变量。此外，MariaDB 服务器重启时，为了再次进行连接复制，需要记录最后执行的二进制日志事件的位置，所以将其记录到 mysql 数据库的 gtid_salve_pos 数据表。

再次讨论使用全局事务 ID 构建复制。由于要使用 GTID 重新构建复制，所以 gtid_slave_pos 与 gtid_current_pos 系统变量全部为空。如下所示，将备份时间点的 GTID 设置给 gtid_slave_pos，然后执行 CHANGE MASTER .. master_use_gitd=slave_pos 命令进行连接复制。

```
MariaDB> SET GLOBAL gtid_slave_pos='0-1-4';

MariaDB> CHANGE MASTER TO master_host="127.0.0.1", master_port=4000,
                          master_user="root", master_use_gtid=slave_pos;
MariaDB> START SLAVE;
MariaDB> SHOW SLAVE STATUS\G
*************************** 1. row ***************************
               Slave_IO_State: Waiting for master to send event
                  Master_Host: 127.0.0.1
                  Master_User: root
                  Master_Port: 4000
                Connect_Retry: 60
              Master_Log_File: binary-log.000002
          Read_Master_Log_Pos: 353
               Relay_Log_File: relay-bin.000002
                Relay_Log_Pos: 641
        Relay_Master_Log_File: binary-log.000002
             Slave_IO_Running: Yes
            Slave_SQL_Running: Yes
...
          Exec_Master_Log_Pos: 353
              Relay_Log_Space: 941
              Until_Condition: None
```

```
              Until_Log_File:
               Until_Log_Pos: 0
...
       Seconds_Behind_Master: 0
Master_SSL_Verify_Server_Cert: No
...
            Master_Server_Id: 1
...
                  Using_Gtid: Slave_Pos
                 Gtid_IO_Pos:
1 row in set (0.00 sec)
```

在 CHANGE MASTER 命令中使用 master_use_gtid 选项后，SHOW SLAVE STATUS 的 Using_Gtid 字段就会出现 Slave_Pos 或 Current_Pos。需要再次连接主服务器时，再次使用该信息。

MariaDB 10.0 内部使用 mysql 数据库的 gtid_slave_pos 数据表保存 GTID 值。存储于 gtid_slave_pos 数据表的值也由 gtid_slave_pos 系统变量管理。因此，MariaDB 5.5 升级到 10.0 后，一定要使用 mysql_upgrade 实用工具生成 MariaDB 10.0 中所需的字典数据表。若想在 MariaDB 服务器异常终止时不损坏并保存 gtid_slave_pos 值，则要使用 InnoDB 或 XtraDB 存储引擎创建 mysql 数据库的 gtid_slave_pos 数据表。请注意，gtid_slave_pos 数据表不能直接使用 UPDATE 或 INSERT 等 DML 语句编辑其中的值。若需要修改 gtid_slave_pos 值，则应该如前示例所示，使用 SET GLOBAL gtid_slave_pos='…'; 命令。

mysql 数据库的 gtid_slave_pos 数据表结构如下。数据表的各列用于存储全局事务 ID 的各个组成部分。sub_id 数据列与全局事务无关，从服务器要处理并行复制时，使用该 id 值。其实，sub_id 数据列与复制的位置信息无关，若从服务器使用 4 个线程进行复制，则 gtid_slave_pos 数据表将产生 4 条记录。从服务器的各线程只将各自的记录更新到 gtid_slave_pos 数据表，最终，从服务器运行的位置由 4 条记录中拥有最后 seq_no 的值确定。这样设计是为了不使以多线程运行的从服务器彼此成为瓶颈。

```
CREATE TABLE gtid_slave_pos (
  domain_id INT UNSIGNED NOT NULL,
  sub_id BIGINT UNSIGNED NOT NULL,
  server_id INT UNSIGNED NOT NULL,
  seq_no BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (domain_id, sub_id)
) ENGINE=InnoDB;

MariaDB [mysql]> SELECT * FROM gtid_slave_pos;
+-----------+--------+-----------+--------+
| domain_id | sub_id | server_id | seq_no |
```

```
+-----------+--------+-----------+--------+
|         0 |      3 |         1 |     14 |
+-----------+--------+-----------+--------+
```

此外，gtid_slave_pos 数据表根据域识别符的不同也拥有不同记录，这在多源复制或者在从服务器中并行处理查询时非常有用。更多相关内容将在多源复制与多线程从服务器中进行详细讲解。

2. 对使用 GTID 的复制进行管理

MariaDB 10.0 提供了几个管理命令，对使用全局事务 ID 的复制进行连接复制或停止等管理。当然，与 MariaDB 5.5 中使用的复制管理命令没有太大不同，但全局事务 ID 的格式与以前不一样，同时，执行命令所需的选项也有一些变化。

借助二进制日志文件名与位置查找全局事务 ID

通常，XtraBackup 与 mysqldump 这类工具仍然不支持 MariaDB 的全局事务 ID。因此，备份时间点的二进制日志往往不是全局事务 ID，而是二进制日志文件名与偏移位置。此时使用 BINLOG_GTID_POS() 函数，利用默认版本的二进制位置信息，可以访问 GTID 值。

```
SELECT BINLOG_GTID_POS('binary-log.000002', 123);
=> '0-1-12'
```

等待全局事务 ID

各位可能在 MySQL 5.5 或 MariaDB 5.5 中使用过 MASTER_POS_WAIT() 函数。以二进制日志文件名与位置作为参数调用 MASTER_POS_WAIT() 函数，从服务器同步到相关时间点时，该函数返回。也就是说，将其用作从服务器同步化的“闹钟”。然而，MariaDB 10.0 开始，可以使用的全局事务 ID 无法使用 MASTER_POS_WAIT 函数。因此，MariaDB 10.0.9 可能引入 MASTER_GTID_WAIT 函数。MASTER_GTID_WAIT 函数是 MASTER_POS_WAIT() 函数的 GTID 版本，但它与 MASTER_POS_WAIT() 函数有一些不同。MariaDB 10.0 的多源复制中，MASTER_POS_WAIT() 函数仅针对一个复制运行。但执行多源复制的从服务器中，MASTER_GTID_WAIT 函数可以对所有复制通道进行等待。MASTER_GTID_WAIT() 函数的第二个参数表示用时，时间为整数时，单位为秒。当然也可以使用小数值，此时可以进行毫秒级或微秒级的超时设置。

```
SELECT MASTER_GTID_WAIT('0-1-12', 10);
```

初始复制连接

使用全局事务 ID 进行连接复制时，首先必须在 gtid_salve_pos 系统变量中设置要连接的时间点的 GTID。执行 CHANGE MASTER 命令时，必须同时使用 master_use_gtid=slave_pos 选项，以使复制连接可以从 gtid_slave_pos 系统变量设置的 GTID 开始。

```
SELECT BINLOG_GTID_POS('binary-log.000002', 123);
SET GLOBAL gtid_slave_pos='0-1-12';
```

```
CHANGE MASTER TO master_host='',master_port='',
  master_user='',master_passsword='',master_use_gtid=slave_pos;
```

停止复制并删除复制相关设置

MariaDB 10.0 中，停止复制或删除所有复制相关元信息的命令与 MariaDB 5.5 中的命令是一样的。

```
STOP SLAVE;
RESET SLAVE ALL;
```

将基于二进制日志文件与位置的复制修改为基于 GTID 的复制

若想将基于二进制日志文件名与位置的复制转换为基于 GTID 的复制，如下所示，使用 CHANGE MASTER 命令时要一起使用 master_use_gtid=current_pos 选项。若主 MariaDB 服务器向二进制日志设置了 GTID，那么即使采用以前的方式（使用二进制日志文件名与位置进行连接），从服务器也会得到主服务器的 GTID 并进入追踪状态。此时，从服务器执行二进制日志事件，继续存储到 gtid_current_pos 系统变量。在 CHANGE MASTER 命令使用 master_use_gtid=current_pos 选项，复制就会从 gtid_current_pos 系统变量设置的 GTID 开始。

```
STOP SLAVE;
CHANGE MASTER TO master_host='',master_port='',
  master_user='',master_passsword='',master_use_gtid=current_pos;
START SLAVE;
```

切换主 MariaDB

主 MariaDB 服务器有时会发生故障，有时也需要按计划切换主从服务器。此情形下，若复制是基于全局事务 ID 进行的，则只要在 CHANGE MASTER 命令的 master_host 选项中输入新主 MariaDB 服务器的主机名，如下所示。这样，MariaDB 服务器就会从新的主 MariaDB 服务器接收以前主服务器中最后执行的二进制日志位置（GTID）之后的事件，然后进行复制同步。使用全局事务 ID 时，将复制的主服务器切换为新的主机后，可以不必另行设置二进制日志的位置或 GTID。

```
STOP SLAVE;
CHANGE MASTER TO master_host='new_master',master_port=3306,master_user='',
master_passsword='';
START SLAVE;
```

运行到指定位置后停止复制

MySQL 5.5 或 MariaDB 5.5 中，可以对复制进行设置，使之只执行到特定位置的二进制日志，比如 START SLAVE UNTIL MASTER_LOG_FILE=mysql-bin.000002.MASTER_LOG_POS=1230。使用全局事务 ID 的情形下，可以如下启动复制，同步到指定的 GTID 时，复制自动停止。

```
STOP SLAVE;
START SLAVE UNTIL master_gtid_pos='0-1-15';
```

注意 MariaDB 10.0 中，与复制有关的命令其实更加复杂。虽然也有对全局事务 ID 的更改，但也添加了多源复制功能，CHANGE MASTER、START SLAVE 与 STOP SLAVE 命令的语法也有所改变。各位可以先不考虑多源复制部分，只记住全局事务环境中复制相关命令的编写方法即可。

3. 在使用 GTID 的从服务器中跳过事务

用户失误或 MariaDB 服务器异常终止会导致从服务器发生如下主键重复错误。

```
Errno: 1062
Error: Error 'Duplicate entry '17' for key 'PRIMARY'' on query.
       Default database: 'test'. Query: 'insert into test.test values (17)'
```

MariaDB 5.5 的基于二进制日志文件与位置进行的复制中，只要使用 SQL_SLAVE_SKIP_COUNTER 系统变量跳过出现问题的事件即可解决上述问题。而使用全局事务 ID 复制环境的从服务器中，无法再使用 SQL_SLAVE_SKIP_COUNTER 系统变量。MariaDB 10.0.7 版本会完全忽略 SQL_SLAVE_SKIP_COUNTER 系统变量的值，而不会出现任何警告或错误信息，目前正处于 BUG 提交状态。将 SQL_SLAVE_SKIP_COUNTER 系统变量设置为 1 时，其实无法跳过一个二进制日志的事件，这种工作方式是有意为之的（不是 BUG）。而将 SQL_SLAVE_SKIP_COUNTER 系统变量设置为大于 1 的值则是错误的，应当针对该情况进行警告或输出错误信息（提交 BUG）。

由此看来，无论怎样，即使使用全局事务 ID 也仍然会出现上述问题。由于无法再使用 SQL_SLAVE_SKIP_COUNTER 系统变量，逐一查看并跳过二进制日志事件也不像之前那么容易了。MariaDB 10.0 开始，为了忽视并跳过二进制日志，要将 gtid_slave_pos 系统变量的值重设为新的起始 GTID 值。

```
MariaDB> SHOW SLAVE STATUS\G
*************************** 1. row ***************************
               Slave_IO_State: Waiting for master to send event
                  Master_Host: 127.0.0.1
                  Master_User: root
                  Master_Port: 4000
                Connect_Retry: 60
              Master_Log_File: binary-log.000002
          Read_Master_Log_Pos: 1744
               Relay_Log_File: relay-bin.000002
                Relay_Log_Pos: 680
        Relay_Master_Log_File: binary-log.000002
             Slave_IO_Running: Yes
            Slave_SQL_Running: No
...
                  Last_Errno: 1062
```

```
                Last_Error: Error 'Duplicate entry '17' for key 'PRIMARY''
 on query. Default database: 'test'. Query: 'insert into test.test values (17)'

....
                 Using_Gtid: Slave_Pos
                Gtid_IO_Pos: 0-1-7

MariaDB> SHOW GLOBAL STATUS VARIABLES LIKE 'gtid_slave_pos';
+------------------+-------+
| Variable_name    | Value |
+------------------+-------+
| gtid_slave_pos   | 0-1-6 |
+------------------+-------+

MariaDB> STOP SLAVE;
MariaDB> SET GLOBAL gtid_slave_pos='0-1-7';
MariaDB> START SLAVE;
```

上述示例中，因发生错误而停止复制时，查看 gtid_slave_pos 系统变量的值可以确认 GTID 值。经查看发现，当前错误发生的点的 GTID 为 0-1-6，为了跳过一个事件组，只要将 GTID 的值重设为 0-1-7（0-1-6 中保持域识别符与服务器识别符不变，只将最后的事件 ID 增加 1）即可。当然，调整 SQL_SLAVE_SKIP_COUNTER 时要停止复制，修改完成后再次启动复制。

提示 我们经常通过调整 SQL_SLAVE_SKIP_COUNTER 系统变量忽略二进制日志事件，其实，在使用事务的 InnoDB 或 XtraDB 中，这样做是相当危险的。若 AutoCommit 模式中，事务只拥有一个查询，以上这种做法不会产生太大问题。但如果一个事务中含有多个 INSERT、UPDATE、DELETE 语句，那么在事务执行过程中借助 SQL_SLAVE_SKIP_COUNTER 系统变量忽略事件的做法就相当危险。

主从服务器采用 ROW 格式的二进制日志进行复制。如下所示，假设有 tb_skip 数据表，因为某种原因导致主从服务器的数据不一致。

```
CREATE TABLE tb_skip (id INT PRIMARY KEY) ENGINE=InnoDB;

MariaDB_Master> SELECT * FROM tb_skip;
+----+
| id |
+----+
|  1 |
|  2 |
|  3 |
+----+

MariaDB_Slave> SELECT * FROM tb_skip;
```

```
+----+
| id |
+----+
|  1 |
|  3 |
+----+
```

此情形下，若主服务器使用下列一个事务分别删除3条记录，由于从服务器无法查找到id=2的记录，故发生复制错误。

```
MariaDB_Master> BEGIN;
MariaDB_Master> DELETE FROM t WHERE id = 1;
MariaDB_Master> DELETE FROM t WHERE id = 2;
MariaDB_Master> DELETE FROM t WHERE id = 3;
MariaDB_Master> COMMIT;
```

查看从服务器错误后，将SQL_SLAVE_SKIP_COUNTER系统变量设置为1，忽略复制错误会怎么样呢?

```
MariaDB_Slave> SHOW SLAVE STATUS\G
*************************** 1. row ***************************
...
Last_SQL_Errno: 1032
Last_SQL_Error: Could not execute Delete_rows event on table test.tb_skip;
 Can't find record in 'tb_skip', Error_code: 1032; handler error HA_ERR_KEY_NOT_FOUND;
the event's master log binary-log.000002, end_log_pos 1892910
...

MariaDB_Slave> STOP SLAVE;
MariaDB_Slave> SET GLOBAL SQL_SLAVE_SKIP_COUNTER=1;
MariaDB_Slave> START SLAVE;
MariaDB_Slave> SELECT * FROM tb_skip;
+----+-----+
| id | pid |
+----+-----+
|  1 |   1 |
|  3 |   3 |
+----+-----+
```

虽然主服务器中的tb_skip数据表的记录已全部删除，但从服务器tb_skip数据表的记录仍然保持原样。ROW格式中GLOBAL SQL_SLAVE_SKIP_COUNTER的工作方式导致忽略整个事务。而使用STATEMENT格式的二进制日志时，也极有可能发生类似问题。

将SQL_SLAVE_SKIP_COUNTER系统变量设置为1后，MySQL或MariaDB不忽略带有一个DML语句的二进制日志事件，而是忽略包含当前事件的事件组。支持事务的数据表中，事务就是一个

事件组；不支持事务的数据表中，各 DML 语句就是事件组。上述示例中，对于 InnoDB/XtraDB 数据表，若一个事务包含多个 DML 查询，则该事务中的所有 DML 语句都将忽略。

8.1.4 MySQL 5.6 全局事务 ID MySQL

MySQL 5.6 的全局事务 ID 与 MariaDB 10.0 的全局事务 ID 在体系与结构上均不同。MySQL 5.6 的全局事务 ID 中，完全不使用我们熟悉的 server_id，而另外引入了服务器 UUID 这一概念。MySQL 5.6 的全局事务 ID 结构如下。

```
"服务器 UUID" : "事务编号"
```

与 MariaDB 10.0 类似，事务编号都是从 1 开始单调增 1 的。服务器 UUID 并不是由用户单独设置的，而是在 MySQL 服务器启动时自动分配的。当然，也可以使用已经分配到的服务器 UUID。MySQL 5.6 服务器启动时会在数据目录中自动生成 auto.cnf 文件，该文件中有个 [auto] 区域，其下存储着当前服务器的 UUID。

```
[auto]
server-uuid= 6c27ed6d-7ee1-11e3-be39-6c626d957cff
```

注意 将使用 XtraBackup 或 Oracle 企业备份得到的数据文件原原本本地应用到新的从服务器（slave）构建时，可能因为失误导致连 auto.cnf 文件也一起应用了。而 MySQL 5.6 的服务器 UUID 比 server-id 限制更加严格。server-id 只在复制组内拥有唯一编号，而 MySQL 5.6 的服务器 UUID 在全局范围中是唯一的。我个人仍然不明白为何 Oracle 不使用原有的 server-id，而重新引入服务器 UUID 这一全新概念。

要在 MySQL 5.6 中使用全局事务 ID，则要满足以下几个条件。

- 激活二进制日志（log-bin 系统变量）
- 设置服务器 ID（server-id 系统变量）
- 从服务器通过复制将数据修改记录到二进制日志（log-slave-updates 系统变量）
- 激活 GTID 一致模式（enforce-gtid-consistency 系统变量）——建议事项
- 激活使用 GTID（gtid-mode 系统变量）

也就是说，为了使用 MySQL 5.6 的全局事务 ID，必须将从服务器执行的所有 DML 与 DDL 记录到二进制日志，并激活 GTID 一致（Consistent）模式。启用 GTID 一致模式后，将无法使用下列两类查询。

- CREATE TABLE ... SELECT ...
- CREATE TEMPORARY TABLE（事务中不可创建临时表）

CREATE TABLE ... SELECT ...语句是 DDL 与 DML 结合而成的事务。事务中包含的 CREATE TEMPORARY TABLE...命令也一样，不允许这些语句组合是因为全局事务 ID 是以一个事务组为单位进行分配的。假设从服务器执行 CREATE TABLE ... SELECT ...命令时创建了数据表，但查询并插入过程中发生错误。这样，MySQL 从服务器就要回滚 CREATE TABLE...

命令，而 DDL 是无法进行回滚操作的。因此，MySQL 5.6 的 enforce-gtid-consistency 激活模式下，就不能使用上述这两种类型的查询。为了解决该问题，MariaDB 10.0 中实现了 CREATE OR REPLACE 这一全新语法。虽然不知道更多内部细节，但仅从用户角度看，两个公司解决问题的方法应该有很大差别。

注意 MySQL 5.6 的 gtid-mode 选项可以设置为 ON 或 OFF，而在内部，gtid-mode 不是布尔型的系统变量，它是枚举（Enumerator）类型，所以不允许设置为 0 或 1 这类整型值。虽然尚未使用，但我们可以将 gtid-mode 系统变量设置为 UPGRADE_STEP_1 与 UPGRADE_STEP_2。

MySQL 5.6 的全局事务 ID 默认处于禁用状态，若不将 gtid-mode 选项设置为 ON，则无法使用全局事务 ID。MariaDB 10.0 默认全局事务 ID 处于激活状态，并且无禁用方法。由此可见，对于全局事务，两个版本的指向有所不同，或者在应用方面有许多不一样的地方。事实上，对于 MySQL 5.6 的全局事务 ID，参与复制的所有 MySQL 服务器必须全部将 gtid-mode 设置为 ON 才能正常参与复制。若想将正在使用 MySQL 5.5 复制方式的 MySQL 服务器切换为使用全局事务 ID，必须同时终止所有参与复制的服务器，并且需要另外的检查时间。

1. 构建使用 GTID 的复制

如前所述，若想在 MySQL 5.6 中使用全局事务 ID，则所有参与复制的 MySQL 服务器的设置文件必须如下设置。若 MySQL 服务器已经处于运行状态，则需要在修改设置后重启 MySQL 服务器。

```
gtid-mode = ON
enforce-gtid-consistency
log-bin = /log/binary-log
server-id = 1
log-slave-updates
```

用户手册中推荐使用 enforce-gtid-consistency 系统变量，若只有 gtid-mode 设置为 ON，而 enforce-gtid-consistency 未做设置，就会引发如下错误，导致 MySQL 服务器无法启动。

```
2013-10-13 11:39:13 17060 [Note] InnoDB: 5.6.15 started; log sequence number 3592460330
2013-10-13 11:39:13 17060 [ERROR] --gtid-mode=ON or UPGRADE_STEP_1 requires
--enforce-gtid-consistency
2013-10-13 11:39:13 17060 [ERROR] Aborting
```

将基于二进制日志文件与位置的复制变更为基于 GTID 的复制

若 MySQL 服务器已经处于服务运行状态，则 gtid-mode 处于激活状态时，若想使复制使用全局事务 ID，只要如下执行 CHANGE MASTER 命令即可。

```
STOP SLAVE;

CHANGE MASTER TO master_host='',master_port=3306,master_user='',master_password='',
                 master_auto_position=1;
```

```
START SLAVE;
```

构建使用 GTID 的新从服务器

若想构建新的从服务器，并使其使用全局事务 ID 参与复制，则要像 MariaDB 中介绍的那样，得到主服务器的备份以构建从服务器。

(1) 原有 MySQL 服务器的数据备份（mysqldump 或 XtraBackup）
(2) 将备份数据恢复到新的从 MySQL 服务器
(3) 确认备份时间点的 GTID
(4) 将新构建的从服务器的 gtid-purged 系统变量变更为步骤 3 中的 GTID
(5) 在新构建的从服务器中执行 CHANGE MASTER 命令

从主 MySQL 服务器接收数据备份时，若使用 mysqldump 实用工具，则必须一起使用 --master-data=2 与 --set-gtid-purged 选项。若不使用这些选项，由于无法在转储结果文件中确认转储结束时间点的 GTID，所以将无法进行连接复制。使用这两个选项后，转储时间点的 GTID 就会记录到转储文件的最上部，如下所示。

```
SET @@GLOBAL.GTID_PURGED='9a511b7b-7059-11e2-9a24-08002762b8af:1-13';
```

此外，在 MySQL 服务器中使用 XtraBackup 工具进行数据备份时，默认在备份目录下生成 xtrabackup_binlog_info 文件。该文件记录着原来二进制日志文件名与偏移位置，以及备份时间点的 GTID。

```
shell> cat xtrabackup_binlog_info
mysql-bin.000002    1232        c777888a-b6df-11e2-a604-080027635ef5:1-4
```

MySQL 5.6 中，有两个与 GTID 相关的系统变量。

gtid_executed：存储记录在二进制日志文件中的所有 GTID 集。

gtid_purged：存储从二进制日志文件中删除的所有 GTID 集。

恢复使用 mysqldump 工具备份的内容时，gtid_executed 与 gtid_purged 系统变量值会自动进行调整，如下所示。

```
MySQL_Slave> SHOW GLOBAL VARIABLES LIKE 'gtid_executed';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| gtid_executed |       |
+---------------+-------+

MySQL_Slave> SHOW GLOBAL VARIABLES LIKE 'gtid_purged';
+---------------+-------+
| Variable_name | Value |
```

```
+---------------+-------+
| gtid_purged   |       |
+---------------+-------+

MySQL_Slave> SOURCE mysqldump.sql;
...

MySQL_Slave> SHOW GLOBAL VARIABLES LIKE 'gtid_executed';
+---------------+-----------------------------------------+
| Variable_name | Value                                   |
+---------------+-----------------------------------------+
| gtid_executed | 9a511b7b-7059-11e2-9a24-08002762b8af:1-13 |
+---------------+-----------------------------------------+

MySQL_Slave> SHOW GLOBAL VARIABLES LIKE 'gtid_purged';
+---------------+-----------------------------------------+
| Variable_name | Value                                   |
+---------------+-----------------------------------------+
| gtid_purged   | 9a511b7b-7059-11e2-9a24-08002762b8af:1-13 |
+---------------+-----------------------------------------+
```

然而，使用 XtraBackup 获得备份时，必须将 gtid_purged 系统变量直接设置为 xtrabackup_binlog_info 文件的 GTID 值。

```
MySQL_Slave> SET GLOBAL gtid_purged="c777888a-b6df-11e2-a604-080027635ef5:1-4";
```

只有 gtid_executed 系统变量设置为空字符串（Empty String）时，才可以修改 gtid_purged 系统变量。也就是说，若 MySQL 服务器曾用作其他 MySQL 服务器的从服务器，则 gtid_executed 与 gtid_purged 系统变量就不为空字符串，而设置为有效字符串。此时，首先要如下所示在从服务器中执行 RESET MASTER 命令，对 gtid_executed 与 gtid_purged 系统变量进行初始化。这样，gtid_executed 与 gtid_purged 系统变量即初始化为空字符串，然后可以使用“SET GLOBAL gtid_purged=”设置从服务器进行复制的起始位置。

```
MySQL_Slave> SHOW GLOBAL VARIABLES LIKE '%gtid%';
+--------------------------+-----------------------------------------+
| Variable_name            | Value                                   |
+--------------------------+-----------------------------------------+
| enforce_gtid_consistency | ON                                      |
| gtid_executed            | 5b9f35ee-94a3-11e3-8c1a-180373fbd40d:1-2 |
| gtid_mode                | ON                                      |
| gtid_owned               |                                         |
| gtid_purged              | 5b9f35ee-94a3-11e3-8c1a-180373fbd40d:1-2 |
+--------------------------+-----------------------------------------+
```

```
MySQL_Slave> RESET MASTER;
Query OK, 0 rows affected (0.08 sec)

MySQL_Slave> SHOW GLOBAL VARIABLES LIKE '%gtid%';
+--------------------------+-------+
| Variable_name            | Value |
+--------------------------+-------+
| enforce_gtid_consistency | ON    |
| gtid_executed            |       |
| gtid_mode                | ON    |
| gtid_owned               |       |
| gtid_purged              |       |
+--------------------------+-------+
```

下面可以使用 mysqldump 或 XtraBackup 恢复数据，并使用全局事务 ID 进行连接复制。最后使用 CHANGE MASTER 命令进行连接复制，如下所示。使用 CHANGE MASTER 命令时，必须同时使用 master_auto_position=1 选项。

```
MySQL_Slave> CHANGE MASTER TO
             master_host='',
             master_user='',
             master_password='',
             master_auto_position=1;
```

执行上述命令后，MySQL 从服务器会引用 gtid_purged 系统变量中的值，从相应时间点开始连接主服务器进行复制，并对数据进行同步。

```
MySQL_Slave> SHOW SLAVE STATUS\G
...
Slave_IO_Running: Yes
Slave_SQL_Running: Yes
...
Retrieved_Gtid_Set: c777888a-b6df-11e2-a604-080027635ef5:5
Executed_Gtid_Set: c777888a-b6df-11e2-a604-080027635ef5:1-5
...
```

注意 如上示例所示，MySQL 5.6 的全局事务中，GTID 的表示法与 MariaDB 10.0 的全局事务 ID 有所不同。在 MySQL 中使用服务器 UUID，而在 MariaDB 中使用域标识符与服务器标识符。MySQL 5.6 的 GTID 中，最后事务编号有时表示为单一整数，但通常会表示为一个范围，比如“1~5”（从事务 1 开始到事务 5 结束）。MySQL 5.6 将这称为“GTID 集（Set）”。

关于 GTID 集将在 Multi-threaded slave 中进行详细讨论。

2. GTID 相关函数

与 MariaDB 一样，MySQL 也添加了许多与 GTID 相关的命令与函数，典型的有 GTID_SUBSET()、GTID_SUBTRACT()、WAIT_UNTIL_SQL_THREAD_AFTER_GTIDS() 函数。由于 GTID_SUBSET()、GTID_SUBTRACT() 函数的用途尚不明确，故此处不进行讲解。借助 WAIT_UNTIL_SQL_THREAD_AFTER_GTIDS() 函数，可以使用基于 GTID 的等待功能，即等待 MySQL 的复制线程（SQL 线程）运行结束至二进制日志的某个位置。调用该函数时，先等待执行到参数指定的 GTID 处，然后复制的 SQL 线程执行到相关的 GTID 事件并返回结果值。此时，WAIT_UNTIL_SQL_THREAD_AFTER_GTIDS() 函数返回的值为整数，表示从函数调用开始到函数返回执行的全局事务个数。

```
MySQL_Slave> SELECT
               WAIT_UNTIL_SQL_THREAD_AFTER_GTIDS('c777888a-b6df-11e2-a604-080027635ef5:1-5');
+-----------------------------------+
| WAIT_UNTIL_SQL_THREAD_AFTER_GTIDS |
+-----------------------------------+
|                                10 |
+-----------------------------------+
```

在 MySQL 5.6 中使用 CHANGE MASTER 命令时，也要添加 master_auto_position 选项，以使用全局事务 ID 进行连接复制。master_auto_position 选项既可以设置为 0，也可以设置为 1。在 CHANGE MASTER 命令中使用 master_log_file 与 master_log_pos 选项时，不能将 master_auto_position 选项设置为 1。也就是说，使用 master_log_file 与 master_log_pos 选项时，不会使用全局事务 ID，而使用 MySQL 5.5 或以前版本的复制方式进行连接。

```
MySQL_Slave> CHANGE MASTER TO master_host='',master_port='',
                              master_user='',master_password='',master_auto_position=1;
```

若将 master_auto_position 设置为 1，则从服务器连接主 MySQL 服务器时，将进行基于全局事务 ID 的连接复制。此时，从服务器会借助全局事务 ID 告知主 MySQL 服务器传送的接收位置和执行位置。为了掌握二进制日志的执行位置以及传送的接收位置，从服务器要访问 gtid_executed 系统变量中设置的 GTID 或 SHOW SLAVE STATUS 命令的 Retrieved_gtid_set 列值。此外，设置复制时，在从服务器中将 gtid_purged 系统变量设置为复制起始位置。将特定 GTID 设置为 gtid_purged 系统变量后，该值也会自动设置到 gtid_executed 系统变量。MySQL 5.6 服务器中，只有将 gtid-mode 设置为 ON 后，才能在 CHANGE MASTER 命令中使用 master_auto_position=1 选项。

3. 跳过 GTID 事件

MySQL 5.6 的全局事务由服务器 UUID 与事务编号组成，其中事务编号绝对不可以遗漏。这是 MySQL 5.6 全局事务管理非常重要的特征，所以 MySQL 5.6 中不能再使用 SQLSLAVE

SKIP_COUNTER 系统变量忽略二进制日志事件。这在 MariaDB 10.0 中也是一样的，但由于 MySQL 5.6 的全局事务不允许遗漏事务编号，所以从服务器中必须强制加入另外的空事务（dummy transaction）。

首先，假设在 MySQL 5.6 中使用全局事务 ID 进行复制。从服务器中，主键重复导致 INSERT 查询失败，复制也随之停止，如下所示。

```
MySQL_Slave> SHOW SLAVE STATUS \G
*************************** 1. row ***************************
               Slave_IO_State: Waiting for master to send event
                  Master_Host: master_mysql_host_ip
                  Master_User: replication_user
                  Master_Port: 3306
                              ...
             Slave_IO_Running: Yes
            Slave_SQL_Running: No
                              ...
                   Last_Errno: 1062
                   Last_Error: Error 'Duplicate entry '2' for key 'PRIMARY''
on query. Default database: 'test'. Query: 'insert into tb_pk values(2)'
                              ...
              Master_Server_Id: 1
                   Master_UUID: 5b9f35ee-94a3-11e3-8c1a-180373fbd40d
                              ...
            Retrieved_Gtid_Set: 5b9f35ee-94a3-11e3-8c1a-180373fbd40d:3-7
             Executed_Gtid_Set: 5b9f35ee-94a3-11e3-8c1a-180373fbd40d:1-6
                 Auto_Position: 1
```

从上述 SHOW SLAVE STATUS 命令的执行结果中可以看到，最后 Auto_Position 的列值为 1。由此可知，当前复制是基于全局事务连接的。当前 SQL 线程中发生主键重复错误（Error 'Duplicate entry '2' for key 'PRIMARY'' on 'query'...）导致复制停止。此外，从 Retrieved_Gtid _Set 列值与 Executed_Gtid_Set 列值看，当前从服务器得到了主 MySQL 服务器的“5b9f35ee-94a3-11e3-8c1a-180373fbd40d:3-7”GTID 集，从服务器中执行的 GTID 集为“1~5”（从 1 号到 5 号）。

注意 为了使从服务器的复制在发生主键重复错误时停止，在从服务器中强制执行 INSERT 语句后，SHOW SLAVE STATUS 的 Executed_Gtid_Set 数据列将与上述示例不同，其值有 2 个以上的 GTID 集，且使用“,”分隔，如下所示。

```
Retrieved_Gtid_Set : 5b9f35ee-94a3-11e3-8c1a-180373fbd40d:3-7
Executed_Gtid_Set  : 0639742e-94a4-11e3-8c1f-180373fbdcfb:1,
                     5b9f35ee-94a3-11e3-8c1a-180373fbd40d:1-6
```

此处，Executed_Gtid_Set 列的“5b9f35ee-94a3-11e3-8c1a-180373fbd40d:1-6”值是记录于获取自主服务器的二进制日志的最终（错误发生之前）GTID 集，“0639742e-94a4-11e3-8c1f-180373fbdcfb:1”是为了引发错误而在从服务器中强制执行的 INSERT 语句的 GTID 集。MySQL 服务器中，即使由一个主服务器与多个从服务器组成，实际在从服务器中也可以执行 INSERT、UPDATE 这类 DML，所以 Retrieved_Gtid_Set 或 Executed_Gtid_Set 列值可以是使用“,”分隔的多个 GTID 集。

此处的“5b9f35ee-94a3-11e3-8c1a-180373fbd40d:7”全局事务的确获取自主服务器，但它实际并不能执行，处于错误状态。此时若想在从服务器中忽略引发错误的“5b9f35ee-94a3-11e3-8c1a-180373fbd40d:7”事务，就要如下所示先停止复制，然后在从服务器中强制创建空（Dummy）事务并加入二进制日志流。

```
## 下一个作业要在从 MySQL 服务器中执行
## 因此，若 MySQL 服务器为 read_only 模式，就需要取得 SUPER 权限

MySQL_Slave> STOP SLAVE;
Query OK, 0 rows affected (0.02 sec)

## 将 gtid_next 变量设置为引发问题的事务 GTID
## "5b9f35ee-94a3-11e3-8c1a-180373fbd40d:7"
MySQL_Slave> SET gtid_next='5b9f35ee-94a3-11e3-8c1a-180373fbd40d:7';
Query OK, 0 rows affected (0.00 sec)

## 创建无任何 DML 的空事务
MySQL_Slave> BEGIN; COMMIT;
Query OK, 0 rows affected (0.00 sec)

## 设置 gtid_next 变量为自动初始化
MySQL_Slave> SET gtid_next='AUTOMATIC';
Query OK, 0 rows affected (0.00 sec)

## 开始复制
MySQL_Slave> START SLAVE;
Query OK, 0 rows affected (0.02 sec)

MySQL_Slave> SHOW SLAVE STATUS\G
*************************** 1. row ***************************
               Slave_IO_State: Waiting for master to send event
                  Master_Host: master_mysql_host_ip
                  Master_User: replication_user
                  Master_Port: 3306
                        ...
             Slave_IO_Running: Yes
```

```
        Slave_SQL_Running: Yes
                      ...
          Last_IO_Errno: 0
          Last_IO_Error:
         Last_SQL_Errno: 0
         Last_SQL_Error:
                      ...
    Retrieved_Gtid_Set: 5b9f35ee-94a3-11e3-8c1a-180373fbd40d:3-7
     Executed_Gtid_Set: 5b9f35ee-94a3-11e3-8c1a-180373fbd40d:1-7
         Auto_Position: 1
```

注意 本示例中，获取自主服务器的 GTID 集为 3~7，执行的 GTID 集为 1~6。由此可以判断，未从服务器获取的 1~2 号 GTID 集也已经得到执行。因为在主服务器 GTID 集 1~2 的时间点接收主服务器的备份，并将备份恢复到从服务器，然后从 3 号事务进行连接复制。

8.2 多源复制 MariaDB

学习多源复制相关内容前，先了解多源复制（Multi-source replication）的定义，通过梳理术语以防混淆。MySQL 5.5 或 MariaDB 5.5 中，由于只能在主服务器执行 INSERT、UPDATE、DELETE 操作，所以无法通过复制确保这些写查询的扩展性。为了对写操作进行扩展，会如图 8-6 所示，连接两台 MySQL 服务器，形成相互复制的关系。这种复制结构中，两台服务器都执行 INSERT、UPDATE、DELETE 查询。虽然这种结构无法对写性能进行很好的扩展，但有些地方仍然会采用这种结构。这种两台服务器都是主服务器的结构经常称为“主服务器 - 主服务器复制”。

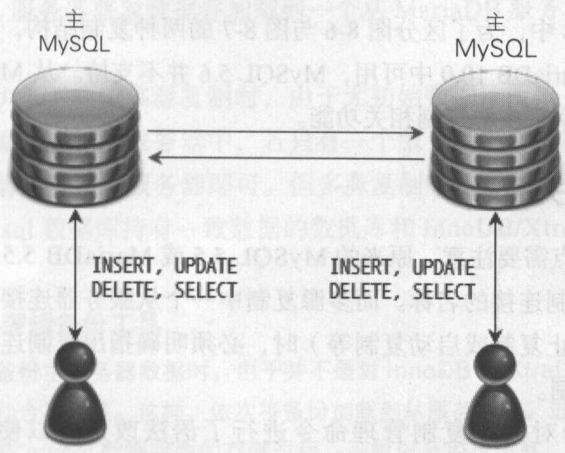


图 8-6 主服务器 – 主服务器复制

MySQL 5.5 或 MariaDB 5.5 的复制中，一台从服务器无法拥有两台以上的主服务器。MySQL 5.5 或 MariaDB 5.5 的复制结构中，绝对不可能的构成就是同时复制两台以上的主服务器。但从 MariaDB 10.0 开始，一台从服务器可以从两台以上的主 MariaDB 服务器复制数据，如图 8-7 所示，这称为“多源（Multi-source）复制”。

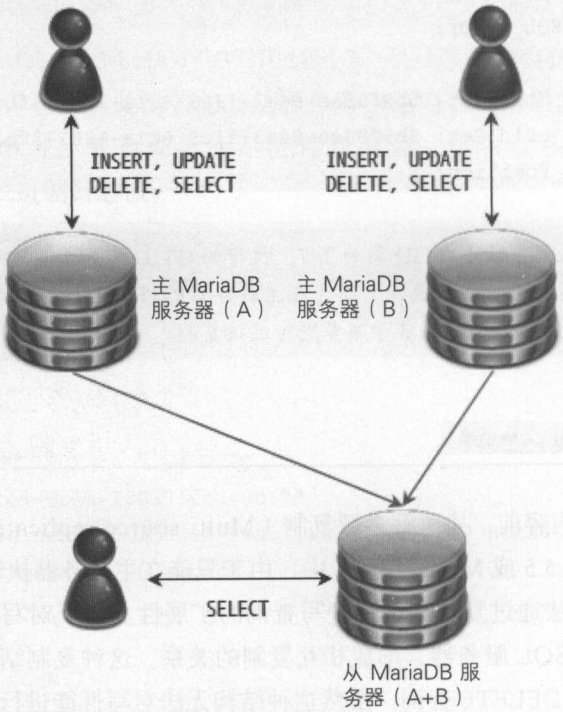


图 8-7 多源复制

由图 8-7 可知，多源复制其实也是从两个主服务器复制数据，故也可以视为“主服务器 - 主服务器复制”。MariaDB 中，为了区分图 8-6 与图 8-7 的两种复制结构，使用“多源复制”这一名称。多源复制仅在 MariaDB 10.0 中可用，MySQL 5.6 并不支持。从 MySQL 5.7 的开发网站或工作日志看，好像正在实现多源复制相关功能。

8.2.1 多源复制相关命令

多源复制中还有一点需要注意，原来的 MySQL 5.5 或 MariaDB 5.5 中，只拥有一个主服务器连接时，无需区分复制连接的名称。而多源复制中一个从服务器连接到多个主服务器，所以使用复制相关命令（停止复制或启动复制等）时，必须明确指出复制连接的名称，以选择要连接哪个主服务器进行复制。

MariaDB 10.0 开始对如下复制管理命令进行了语法改进，以使所有命令都可以使用 connection_name 选项。管理复制的命令中，connection_name 选项是可选项。若复制管理命令中未使用 connection_name 选项，则 MariaDB 服务器会使用 default_master_connection 系统变量

（会话变量）中设置的复制连接名称。default_master_connection 系统变量的初始值为空字符串（""），表示不使用多源复制，适用于只从一个主服务器进行复制的情形，就像 MariaDB 5.5 中一样。也就是说，不使用多源复制时，若不将 default_master_connection 系统变量另外设置为特定值，则可以像 MariaDB 5.5 中那样，不使用 connection_name 选项进行所有复制相关工作。

- CHANGE MASTER["connection_name"]...
- FLUSH RELAY LOGS["connection_name"]
- MASTER_POS_WAIT(....,["connection_name"])
- RESET SLAVE["connection_name"]
- SHOW RELAYLOG["connection_name"]EVENTS
- SHOW SLAVE["connection_name"]STATUS
- START SLAVE["connection_name" ...]]
- STOP SLAVE["connection_name"]...

MariaDB 5.5 开始就在使用上述复制相关命令，故不再详细说明。以上命令中，"[]" 表示可选项，可以省略不用。此外，MariaDB 10.0 中新引入了下列命令，进行多源复制时，使用这些命令可以同时访问所有复制连接的状态，以及启动或停止所有复制。

- SHOW ALL SLAVES STATUS
- START ALL SLAVES...
- STOP ALL SLAVES...

8.2.2 构建多源复制

事实上，多源复制与原 MariaDB 5.5 或 MySQL 5.5 中的复制并无太大差异。只在进行连接复制时，要将特定时间点的数据传送到从服务器，该部分工作略有些棘手。也就是说，从两台彼此不同的主 MySQL 服务器获取数据并加载到一个从 MariaDB 服务器中，是一件复杂且棘手的事情。

在无任何数据的状态下构建多源复制时，由于无初始数据，所以只要连接并进行多源复制即可。此外，连接多源复制的主服务器中，若只有一个服务器拥有数据，则只要将拥有数据的主服务器中备份的数据恢复到从服务器即可。但多源复制中，若要从两个以上的主服务器获取数据，就要考虑与 mysql 数据库持有一致数据的数据库和 InnoDB/XtraDB 的系统表空间的冲突与合并。

使用 mysqldump 逻辑备份工具

使用 mysqldump 备份主服务器数据时，由于并不是对 InnoDB 或 XtraDB 的系统表空间进行物理备份，故不会出现这种合并问题。这样，依次将备份加载到从服务器中，进行复制连接将不会出现什么问题。当然，存储在 mysql 数据库中的存储过程、函数以及用户信息、权限相关的数据表可能出现重复，但由于这些数据表中的记录并不多，可以手工调整。但如果备份的数据非常大，那么使用 mysqldump 备份或加载都将花费相当长的时间。

使用 XtraBackup 物理备份工具

使用 XtraBackup 这类物理备份工具时，可以将大容量数据库快速备份到从服务器。对于 InnoDB/XtraDB 存储引擎数据表，除了 MariaDB 的 FRM 文件外，存储引擎自身的系统表空间也保存着数据表的信息。使用 XtraBakcup 这类物理备份工具进行备份时，将会原样复制系统表空间以及 MariaDB 服务器中的所有数据文件，然后再使用这些备份进行恢复。此时，无法将从两台主服务器获取的系统表空间合并为一个。

从多台服务器获取数据并顺利加载到从服务器的方法是，适当混用 mysqldump 与 XtraBackup 两种工具。假设有两台主服务器 A 与 B，从服务器 C 连接它们复制数据，然后根据服务器 A 与 B 的数据大小适当混用两种工具，按一定顺序恢复即可。

服务器 A 与 B 数据量不大时

如果两台主服务器拥有的数据量不大，并且可以使用 mysqldump 工具进行备份，同时在指定时间内能够于从服务器中完成数据的恢复工作，那么只要将从服务器 A 与 B 进行 mysqldump 备份的结果依次加载到 C 从服务器，再检查存储进程、函数以及用户信息与权限即可。

服务器 A 拥有的数据量大而服务器 B 拥有非常少的数据时

若服务器 A 与 B 拥有的数据量差别很大，那么使用 mysqldump 备份数据量少的服务器，使用 XtraBackup 备份数据量大的服务器即可。由于服务器 A 拥有的数据量很大，所以首先使用 XtraBackup 工具备份服务器 A 的数据，并恢复到从服务器 C。然后使用 mysqldump 工具备份服务器 B 的数据，并加载到从服务器 C。当然，此时最好也检查存储进程、函数以及用户信息与权限信息。

服务器 A 与 B 都拥有大量数据时

主服务器 A 与 B 都拥有大量数据时，要使用 XtraBackup 工具对其进行物理备份，并将数据表数较多的一方先恢复到从服务器 C。由于 InnoDB/XtraDB 的系统表空间冲突，其余备份将无法使用 XtraBackup 的 copy-back 命令进行恢复。因此，针对其余备份，先使用 InnoDB 的导出命令导出 InnoDB/XtraDB 数据表的 ibd 文件，再将其导入从服务器 C。由于要采用手动方式逐个表格导入 / 导出表空间，所以尽可能在对数据表较少的服务器进行备份时使用该方法。

假设现在从服务器 C 已经由主服务器 A 与 B 获取了所有数据，复制（备份）服务器 A 与 B 数据时间点的二进制日志分别如下所示。下面进行连接多源复制。

```
服务器 A 二进制日志位置：binary-log.000002 : 100
服务器 B 二进制日志位置：binary-log.000011 : 200
```

只要知道了要开始复制的二进制日志位置，然后使用 MariaDB 5.5 中的 CHANGE MASTER 命令与 START SLAVE 命令开始连接复制即可。但请注意，CHANGE MASTER 与 START SLAVE 命令以各复制连接为单位分配 connection_name。

```
--// 与主服务器 A 进行连接复制
MariaDB_Slave> CHANGE MASTER 'replication_A' TO master_host='hostname_A',
                    master_user='replication_user', master_password='
replication_pass',
```

```
                        master_port=3306,
                        master_log_file='binary-log.000002', master_log_pos=100;

--// 与主服务器 B 进行连接复制
MariaDB_Slave> CHANGE MASTER 'replication_B' TO master_host='hostname_B',
                        master_user='replication_user', master_password='
replication_pass',
                        master_port=3306,
                        master_log_file='binary-log.000011', master_log_pos=200;

--// 依次启动主服务器 A 与 B 的复制
MariaDB_Slave> START SLAVE "replication_A";
MariaDB_Slave> START SLAVE "replication_B";

MariaDB_Slave> SHOW ALL SLAVES STATUS\G
```

若认为每次都要写出 connectionname 命令很麻烦，则可以如下所示，将 default_master connection 系统变量（会话变量）设置为 connection_name，然后即可像在 MariaDB 5.5 中那样使用 CHANGE MASTER 与 START SLAVE 等命令。

```
--// 与主服务器 A 进行连接复制
MariaDB_Slave> SET SESSION default_master_connection="replication_A";
MariaDB_Slave> CHANGE MASTER TO master_host='hostname_A',
                        master_user='replication_user', master_password='
replication_pass',
                        master_port=3306,
                        master_log_file='binary-log.000002', master_log_pos=100;
MariaDB_Slave> START SLAVE;

--// 与主服务器 B 进行连接复制
MariaDB_Slave> SET SESSION default_master_connection="replication_B";
MariaDB_Slave> CHANGE MASTER TO master_host='hostname_B',
                        master_user='replication_user', master_password='
replication_pass',
                        master_port=3306,
                        master_log_file='binary-log.000002', master_log_pos=100;
MariaDB_Slave> START SLAVE;
```

注意 与 MariaDB 5.5 中的复制一样，MariaDB 10.0 的多源复制中，参与复制的所有服务器的 server-id 也必须是唯一的，请各位务必牢记。此外，也一定要检查全局事务 ID 中的 domain-id，查看它们是否彼此不重复。

8.2.3 多源复制与全局事务

使用全局事务ID的MariaDB中也可以使用多源复制，还可以使用mysqldump或XtraBackup等工具，将多个主服务器的数据复制到从服务器，此时使用8.2.2节讲解的方法即可。使用BINLOG_GTID_POS()函数通过二进制日志文件名与偏移位置获取GTID的方法如前所述，请参考全局事务ID相关内容。

下面学习在从服务器C中连接复制主服务器A与B数据的方法。假设从服务器C已经获取了主服务器A与B中的数据，复制（备份）服务器A与B数据时间点的二进制日志中，GTID分别如下所示。下面进行连接多源复制。

```
服务器A的二进制日志GTID: 1-1-2
服务器B的二进制日志GTID: 2-1-2
```

首先查看从服务器C中与GTID相关的系统变量值是否如下被初始化。若这些系统变量仍然保留着以前使用过的污染值，则可以使用RESET MASTER命令对它们进行初始化，相关内容请参考全局事务ID的有关部分。

```
MariaDB_Slave>SHOW GLOBAL VARIABLES LIKE 'gtid%';
+------------------+-------+
| Variable_name    | Value |
+------------------+-------+
| gtid_binlog_pos  |       |
| gtid_binlog_state|       |
| gtid_current_pos |       |
| gtid_domain_id   | 0     |
| gtid_slave_pos   |       |
| gtid_strict_mode | OFF   |
+------------------+-------+
```

连接使用GTID的复制之前，首先要将复制同步开始的GTID值设置到gtid_slave_pos系统变量。前面已经讲解过只有一台主服务器的情形，而像多源复制一样，拥有多台主服务器而设置gtid_slave_pos系统变量时，多个主服务器的GTID之间要使用“,”分隔，如下所示。

```
MariaDB_Slave>SET GLOBAL gtid_slave_pos='2-1-2,1-1-1';

MariaDB_Slave>SHOW GLOBAL VARIABLES LIKE 'gtid%';
+------------------+------------+
| Variable_name    | Value      |
+------------------+------------+
| gtid_binlog_pos  |            |
| gtid_binlog_state|            |
| gtid_current_pos | 1-1-1,2-1-2|
| gtid_domain_id   | 0          |
```

```
| gtid_slave_pos     | 1-1-1,2-1-2 |
| gtid_strict_mode   | OFF         |
+--------------------+-------------+
```

设置 gtid_slave_pos 系统变量时，服务器 A 与 B 的 GTID 值先后顺序并不重要，只要使用“,”分隔各主服务器的 GTID 值即可。

```
--// 与主服务器 A 进行连接复制
MariaDB_Slave> CHANGE MASTER 'replication_A' TO
                  master_host='hostname_A', master_port=3306,
                  master_user='replication_user', master_password='
replication_pass',
                  master_use_gtid=slave_pos;

--// 与主服务器 B 进行连接复制
MariaDB_Slave> CHANGE MASTER 'replication_B' TO
                  master_host='hostname_B', master_port=3306,
                  master_user='replication_user', master_password='
replication_pass',
                  master_use_gtid=slave_pos;

--// 开始与主服务器 A、B 进行复制同步
MariaDB_Slave> START SLAVE 'replication_A';
MariaDB_Slave> START SLAVE 'replication_B';

MariaDB_Slave> SHOW SLAVE 'replication_A' STATUS \G
*************************** 1. row ***************************
               Slave_IO_State: Waiting for master to send event
                  Master_Host: hostname_A
                  Master_User: replication_user
                  Master_Port: 3306
                              ...
             Master_Log_File: mysql-binary.000001
         Read_Master_Log_Pos: 589
              Relay_Log_File: mysql-relay-replication_A.000002
               Relay_Log_Pos: 798
       Relay_Master_Log_File: mysql-binary.000001
            Slave_IO_Running: Yes
           Slave_SQL_Running: Yes
                              ...
                  Using_Gtid: Slave_Pos
                 Gtid_IO_Pos: 1-1-2,2-1-2
```

使用全局事务 ID 时，也可以使用 default_master_connection 系统变量，这样就不必每次在各复制管理命令中输入 connection_name。

8.3 多线程复制 MariaDB MySQL

MariaDB 10.0 与 MySQL 5.6 都提供了基于多线程的复制功能，但就像全局事务 ID 一样，它们在实现时遵从了不同的“哲学”，所以工作方式与使用方法略有不同。MySQL 中使用服务器 UUID，而 MariaDB 中使用域识别符与服务器识别符。MySQL 5.6 的 GTID 中，最后事务的编号有时为单一整数，但通常是一个范围，比如 1~5（从事务 1 到事务 5）。

MySQL 5.6 将其称为 GTID 集，而 MariaDB 中不会使用这种 GTID 集，总是为一个事务编号。为什么会有这种差异呢？下面通过学习并比较 MySQL 5.6 与 MariaDB 10.0 的多线程复制一探究竟。

8.3.1 MySQL 5.6 的多线程复制

MySQL 5.6 的多线程复制不仅可以用于 MySQL 5.5 这种不使用全局事务 ID 的复制环境，也可以用于 MySQL 5.6 这种使用全局事务 ID 的环境。MySQL 5.6 的多线程复制支持以数据库为单位的并行处理。也就是说，若 MySQL 5.6 服务器中只有一个数据库，则使用多线程复制就没有什么优势。建议各位根据一个 MySQL 服务器中数据库的数量设置相应数量的线程。图 8-8 表示 MySQL 5.6 中以数据库为单位的多线程工作方式。协调者（Coordinator）读取中继日志（Rely log）文件的二进制日志事件，然后以数据库为单位对事件进行分类，再分发到各 SQL 线程执行。

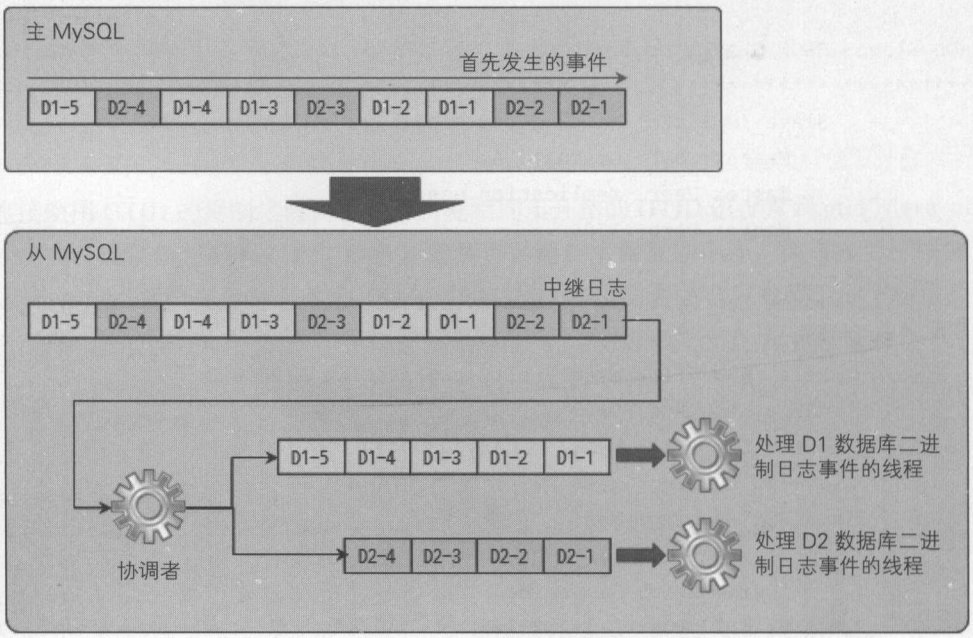


图 8-8 MySQL 5.6 多线程从服务器的工作方式

协调者并非只以数据库为单位、将二进制日志事件分发到各 SQL 线程。假设有 3 个 SQL 线程，并有 DB1、DB2、DB3 三个数据库，如下处理二进制日志。第一个与第二个事务只使用数据库 DB1 与 DB2，而第三个事务先更新 DB3，再更新 DB1。

此时，如图 8-9 所示，协调者先通知第一个 SQL 线程处理 UPDATE DB1 事件，随后立即通知第二个 SQL 线程处理 UPDATE2 事件。由于第三个事务的第一条 DML 语句（UPDATE DB3）与第一个 SQL 线程执行的 DB1 及第二个 SQL 线程执行的 DB2 彼此不冲突，所以立刻通知第三个 SQL 线程执行。要执行第三个事务的第二条 DML 语句（UPDATE DB1）时，由于该语句操作的数据库与第一个 SQL 线程正在执行的 UPDATE DB1 语句涉及的数据库相同，故不会将该语句传递给第三个 SQL 线程，而要一直等待，直到第一个线程执行完 UPDATE DB1 语句。当然，事务 1 中的 UPDATE DB1 与事务 3 中的 UPDATE DB1 既可能是修改同一数据表同一条记录的查询，也可能只是数据库相同而数据表或记录不同。但 MySQL 5.6 的多线程从服务器不会考虑表级或记录级别的冲突，只比较数据库是否相同，然后执行多线程处理。

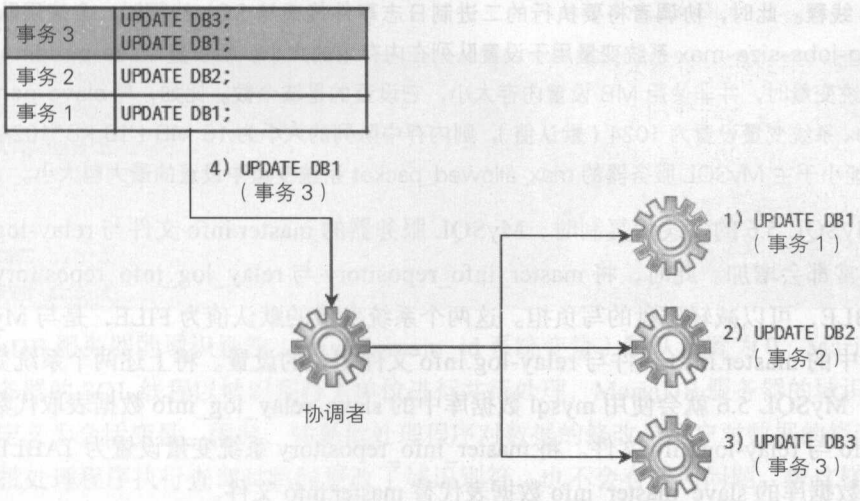


图 8-9 MySQL 5.6 多线程从服务器的协调者工作方式

在 MySQL 5.6 中使用多线程从服务器时，只要将 slave_parallel_workers 系统变量设置为要使用的从服务器 SQL 线程数即可。slave_parallel_workers 系统变量允许的取值范围为 0~1024，设置为 0 表示不使用多线程从服务器，而采用与 MySQL 5.5 一样的单线程从服务器模式进行复制。将 slave_parallel_workers 系统变量设置为 1 与 0 是有差别的，slave_parallel_workers 值为 1 时，多线程从服务器的代码块（多个从服务器线程间的同步或协调作业）都会执行，多线程从服务器可能发生过载，但实际复制由单一线程负责执行。因此，建议各位不要将 slave_parallel_workers 设置为 1[①]。除 slave_parallel_workers 系统变量外，还有下列几个与多线程从服务器相关的系统变量。

① 该内容基于本书编写时的最新 MySQL 版本 5.6.15，更新版本中可能会对此进行改进，详细内容请参考新版本发布说明。

slave-checkpoint-period

从服务器执行的二进制日志事件的位置必须记录到 SHOW SLAVE STATUS 命令的结果或者存储中继日志信息的文件（relay-log.info）。此时，若频繁将该信息写入磁盘，则会大大增加服务器的负荷，所以建议以合适的周期进行更新。slave-checkpoint-period 系统变量以毫秒为单位进行设置，依据该变量设置的周期，中继日志信息文件或 SHOW SLAVE STATUS 命令的结果值会更新为当前状态值。

slave-checkpoint-group

与 slave-checkpoint-period 系统变量具有相同功能，但 slave-checkpoint-group 系统变量设置的是事务的个数而非时间。比如，将 slave-checkpoint-group 设置为 10，则从服务器执行完 10 个事务后，更新中继日志信息文件（relay-log.info）与 SHOW SLAVE STATUS 使用的二进制日志事件信息。

slave-pending-jobs-size-max

多线程从服务器中，协调者首先读取二进制日志事件，然后将相关作业传递给负责实际处理的从服务器 SQL 线程。此时，协调者将要执行的二进制日志事件传递给 SQL 线程时，会使用队列。slave-pending-jobs-size-max 系统变量用于设置队列在内存中的大小。但设置 slave-pending-jobs-size-max 系统变量时，并非使用 MB 设置内存大小，它设置的是块个数。比如，将 slave-pending-jobs-size-max 系统变量设置为 1024（默认值），则内存中队列的大小为 16 MB（16 KB*1024）。请注意，该值不能小于主 MySQL 服务器的 max_allowed_packet 系统变量中设置的最大包大小。

使用 MySQL 5.6 的多线程复制时，MySQL 服务器的 master.info 文件与 relay-log.info 文件的写负荷通常都会增加。此时，将 master_info_repository 与 relay_log_info_repository 系统变量设置为 TABLE，可以减轻磁盘的写负担。这两个系统变量的默认值为 FILE，是与 MySQL 服务器数据目录中的 master.info 文件与 relay-log.info 文件相关的设置。将上述两个系统变量设置为 TABLE 后，MySQL 5.6 就会使用 mysql 数据库中的 slave_relay_log_info 数据表取代数据目录中的 master.info 与 relay-log.info 文件。将 master_info_repository 系统变量设置为 TABLE 后，就会使用 mysql 数据库的 slave_master_info 数据表代替 master.info 文件。

MySQL 5.6 中，对于一个主服务器中发生的二进制日志，从服务器会使用多个线程执行。其实，从服务器启动多个线程执行二进制日志文件中记录的事件时，并不按照事件的记录顺序执行。图 8-8 中，若 D2 数据库的二进制日志很小且执行速度快，则 D2 数据库的事件就会快速复制至从服务器，而 D1 数据库事件的复制就会很慢。其实，从服务器进行复制时，并非依据事件在主服务器中的发生顺序，所以 MySQL 5.6 中无法使用 1 个 GTID 值表示从服务器复制到当前时间点的二进制日志位置。因此，MySQL 5.6 引入了“GTID 集”这一概念，MySQL 5.6 复制中的大部分均使用 GTID 集表示。通过 GTID 集可以表示执行到现在的所有全局事务的范围，格式为“服务器 UUID: 起始事务号 – 结束事务号”。

然而，GTID 集发生编号遗漏时，为了标记这样的 GTID 集，需要包含相当多的范围。比如，到现在为止执行了 1~100 号事务，若其中第 10 号、20 号、30 号事务遗漏，那么 GTID 集就应该表示为“UUID:1-9:,UUID:11-19,UUID:21-29,UUID:31-100”。若遗漏的事务编号很多，应

该如何保存这么多的值呢？其实并没有什么很好的方法。因此，MySQL 5.6 的全局事务中不允许出现遗漏的事务编号。

8.3.2 MariaDB 10.0 的多线程复制

MariaDB 10.0 也支持多线程复制，但工作方式与 MySQL 5.6 略有不同。如前所述，MySQL 5.6 中，从服务器的多线程自动以数据库为单位进行工作，这意味着 MySQL 服务器的管理员或开发者不需通过手工操作执行并行处理。但 MariaDB 10.0 中，除多源复制外，多线程复制不会自动运行。MariaDB 10.0 的多线程复制通常以域识别符（gtid_domain_id 系统变量）为基准进行工作。MariaDB 10.0 的复制中，多线程也以数据库为单位运行。

多源复制中，由于各主 MariaDB 服务器的域识别符彼此不同，所以从服务器中，各线程分配到多个主服务器传递来的二进制日志，以处理二进制日志事件。那么，只有 1 个主服务器时，从服务器如何并行处理复制呢？假设 1 个主 MariaDB 服务器中，批处理程序与用户同时进行连接并修改相应数据，如下所示。

```
批处理程序
    INSERT INTO tab_b ..
    UPDATE tab_b ..

用户
    INSERT tab_a ..
    UPDATE tab_a ..
```

MariaDB 服务器的域识别符（gtid_domain_id 系统变量）默认设置为 0。MariaDB 服务器中，从服务器的 SQL 线程以域识别符为单位进行并行处理，MariaDB 服务器的域识别符与全局变量一起定义为会话变量。因此，如果批处理程序对数据的修改与用户对数据的修改完全无关，那么即使批处理程序执行查询时临时更改了域识别符，也不会有任何问题。像这样，更改域标识符之后再执行查询时，从服务器就会根据不同的域标识符使用各线程并行处理查询，如图 8-10 所示。

MariaDB 10.0 的多线程从服务器既可用于多源复制，也可以在需要大量时间修改大量数据时，保证从服务器不会发生复制延迟现象。当然，MariaDB 10.0 的多线程复制是手动的，这与 MySQL 5.6 不同，用户或应用程序可以决定从服务器并行处理各查询还是按照顺序执行。

MySQL 5.6 与 MariaDB 10.0 的多线程复制各有优劣，至于哪个更好，不同开发者或服务特点会形成不同判断结果。但可以确定的是，以数据库为单位的多线程复制可以实现自动化，而以记录为单位的多线程复制——就目前而言——依然相当困难。记录级别的多线程复制最终只能由用户应用程序根据当时的业务逻辑决定，而自动多线程复制并非为了满足这些条件，类似 MariaDB 10.0 的手动多线程复制（用户决定是否使用多线程复制）很有可能成为必需功能。

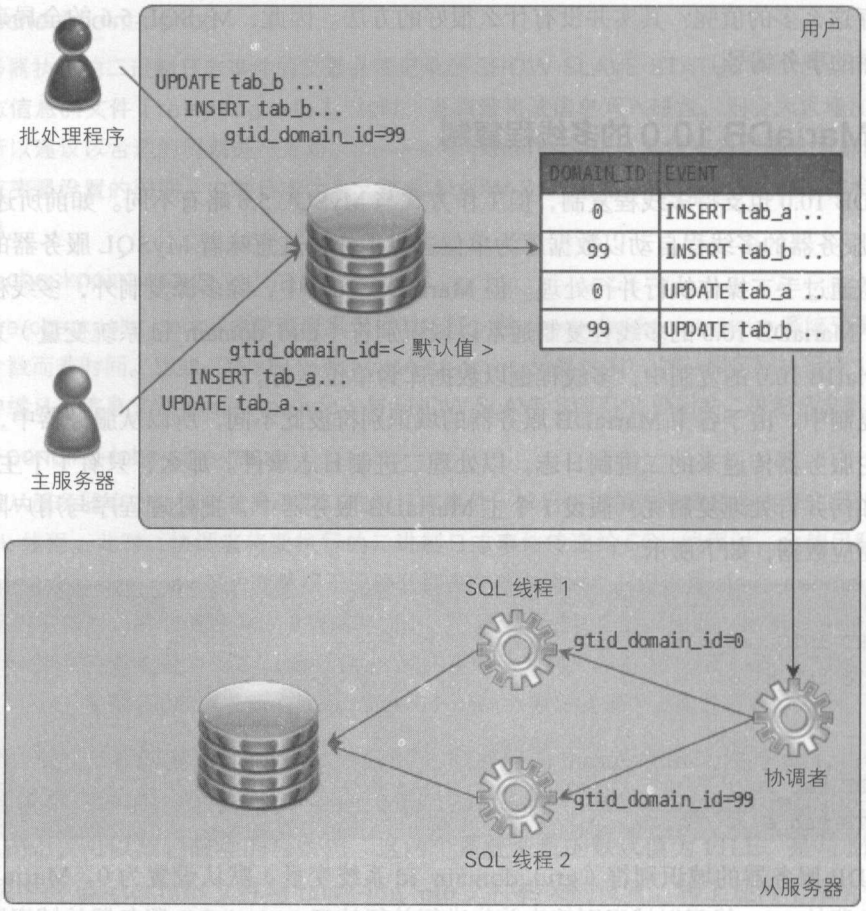


图 8-10 MariaDB 10.0 的多线程从服务器

注意 MariaDB 10.0 的多线程复制必须使用域识别符，但只有全局事务 ID 激活且使用全局事务 ID 进行复制时，域识别符（gtid_domain_id）才有意义。也就是说，MariaDB 10.0 的多线程复制只在使用全局事务时才可用。而 MySQL 5.6 的多线程复制与是否使用全局事务 ID 无关，它使用两个以上的数据库，只要查询以数据库为单位执行，即可使用多线程复制。

MariaDB 10.0 的多线程复制中，将 slave-parallel-threads 系统变量设置为非 0 的正整数，相应数量的从服务器 SQL 线程就会运行。由于 slave-parallel-threads 系统变量是动态变量，所以可以随时调整从服务器的线程个数。但修改 slave-parallel-threads 之前，必须首先执行 STOP SLAVE 命令。修改完成后再执行 START SLAVE 命令重启复制，这样新设置的从服务器线程数才会生效。目前，MariaDB 10.0 中可以使用多线程进行复制的查询有如下几类。

- 主服务器中属于同一提交组的查询
- 拥有不同域识别符的查询
- 多源复制中获取自不同主服务器的查询

MariaDB 中，必须为多线程从服务器在设置文件（my.cnf）内设置 slave-parallel-max-queued 系统变量。只有 slave-parallel-threads 系统变量设为大于 0 的整数时，它才起作用。MariaDB 的多线程从服务器激活后，MariaDB 才会将二进制日志事件提前存储到内存队列，以检查是否可以使用多线程执行二进制事件。slave-parallel-max-queued 系统变量决定为队列分配的内存大小。使用彼此不同的域识别符与全局事务 ID 的复制环境中，该内存队列是必须的。

假设域识别符为 1 的事务与域识别符为 2 的事务连续执行，从服务器中的特定从服务器线程执行域识别符为 1 的查询。此时只有读取并分析下个二进制日志事件，才能知道该事务的域识别符为 2，并可以在其他从服务器线程中并行运行。只有域识别符为 1 的事务的大小比 slave_parallel_max_queued 系统变量的值小，域识别符为 2 的查询才能执行。也就是说，一个事务包含的 SQL 语句或记录信息非常多且大时，必须将 slave_parallel_max_queued 设置得大一些，这样才能在从服务器中使用多线程进行正常处理。

8.4 Crash safe slave MariaDB MySQL

MySQL 与 MariaDB 的复制中，IO 线程与 SQL 线程协同合作，与主服务器进行数据同步，如图 8-11 所示。IO 线程首先通过网络从主服务器获取二进制日志事件，然后以文件形式保存到从服务器的本地磁盘。SQL 线程负责在实际的 MySQL 服务器中重新执行（Replay）IO 线程获取的二进制日志事件。该过程中，IO 线程获取二进制日志的进度信息保存于 master.info 文件，SQL 线程重新执行查询的进度信息保存于 relay-log.info 文件。MySQL 服务器终止并重启运行后，会浏览这些文件，判断应该从哪个时间点再次开始。

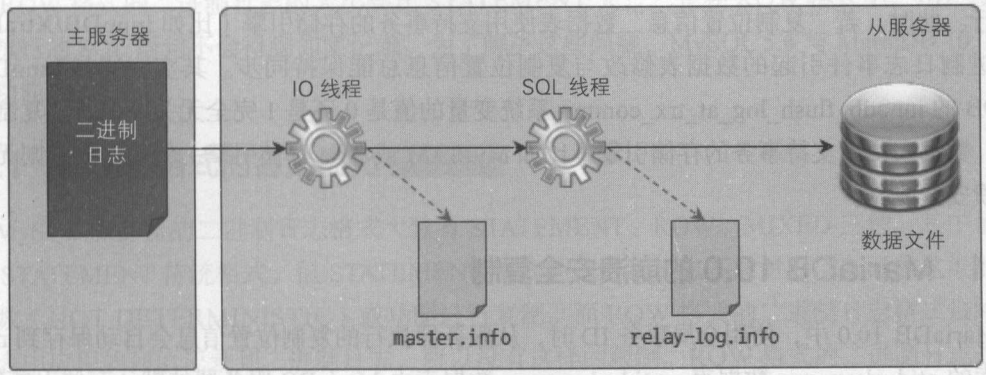


图 8-11 master.info 文件与 relay-log.info 文件

每次逐个复制二进制日志事件时，都要修改这些文件中的信息并同步到磁盘，期间需要大量磁盘操作。因此，通常采用异步方式处理对这些文件的修改。也就是说，虽然每次都要记录到文件，但磁盘的同步会交给操作系统执行（有时会先收集几个二进制日志事件然后一次性写入磁盘）。内存与磁盘数据不同时，若操作系统异常终止，内存中的信息就无法同步到磁盘，这

些信息就会丢失。此时重启 MySQL 服务器，由于 master.info 或 relay-log.info 文件中的信息仍然是旧的，所以会从已经执行的部分重新执行，该过程中经常出现的典型错误是“Duplicate key”。

MariaDB 10.0 与 MySQL 5.6 中，为了防止出现该问题而进行了改善，将 master.info 文件与 relay-log.info 文件的内容保存到 MySQL 服务器的数据表而非磁盘文件。该功能在使用手册中名为“崩溃安全复制”（Crash-safe relication）。MariaDB 10.0 与 MySQL 5.6 在实现崩溃安全复制功能时使用的方法略有不同，但基本原理是一样的。下面简单介绍“崩溃安全复制”的原理，以及 MariaDB 与 MySQL 实现该功能的差异。

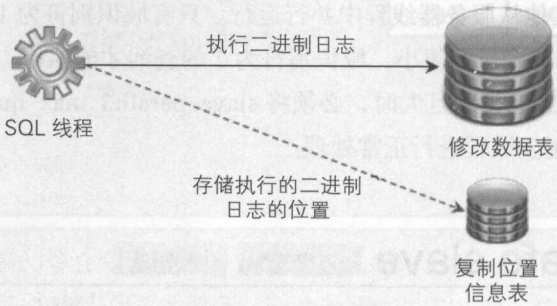


图 8-12 崩溃安全复制结构

图 8-11 展示了使用文件记录二进制日志位置信息的方法。图 8-11 的数据库中，执行二进制日志事件与更新 master.info 或 relay-log.info 文件中的位置信息都不是原子操作（Atomic）。因此，很有可能出现内存与磁盘中的值不一致的情形，从而导致复制从服务器产生同步问题。而崩溃安全复制中，执行的二进制日志事件的位置信息保存到数据表，如图 8-12 所示。也就是说，通过二进制日志事件修改数据的操作与“复制位置信息”数据表的位置修改被作为一个事务执行。此时，若“复制位置信息”数据表使用支持事务的存储引擎（比如 InnoDB/XtraDB），则二进制日志事件引起的数据表修改与复制位置信息总能保持同步。其实，这与 InnoDB 或 XtraDB 的 innodb_flush_log_at_trx_commit 系统变量的值是 0 还是 1 完全无关。但若“复制位置信息”数据表由不支持事务的存储引擎（比如 MyISAM 或 Aria 存储引擎）创建而成，则无法保证崩溃安全复制。

8.4.1 MariaDB 10.0 的崩溃安全复制

MariaDB 10.0 中，使用全局事务 ID 时，从服务器执行的复制位置信息会自动保存到 mysql 数据库的 gtid_slave_pos 数据表，gtid_slave_pos 数据表由 MariaDB 服务器的默认存储引擎创建。也就是说，MariaDB 服务器的默认存储引擎若设置为 MyISAM，则崩溃安全复制功能无法正常运行。

若 MariaDB 10.0 中不使用全局事务，则无法启用崩溃安全复制功能。因为只有激活全局事务 ID 并在复制中应用，才能使用 gtid_slave_pos 数据表。关于 gtid_slave_pos 数据表的详细内容，请参考全局事务 ID 相关章节。

8.4.2 MySQL 5.6 的崩溃安全复制

使用 MySQL 5.6 的崩溃安全复制功能与全局事务 ID 无关。下列两个系统变量用于决定是否使用崩溃安全复制功能。

master-info-repository

master-info-repository 系统变量可以设置为 TABLE 或 FILE，设置为 FILE 时，获取自主服务器的二进制日志文件的信息会记录到磁盘的 master.info 文件；若设置为 TABLE，则激活崩溃安全复制功能。此时，二进制日志文件的信息会存储到 mysql 数据库的 slave_master_info 数据表，而非 master.info 文件。该系统变量默认设置为 FILE。

relay-log-info-repository

与 master-info-repository 系统变量一样，该变量也可以设置为 TABLE 或 FILE。若设置为 FILE，则 SQL 线程会将从服务器中执行二进制日志事件的进度相关信息记录到磁盘的 relay-log.info 文件；若设置为 TABLE，则记录到 mysql 数据库的 salve_relay_log_info 数据表。该变量的默认值也为 FILE。

master-info-repository 与 relay-log-info-repository 系统变量全部设置为 TABLE 时，MySQL 5.6 服务器会自动将“更新二进制文件的位置到数据表”的操作与执行二进制日志事件作为一个事务进行处理。也就是说，将两个系统变量设置为 TABLE 后，随机激活崩溃安全复制功能。

8.5 改善基于 ROW 的复制功能 MariaDB MySQL

已经在复制中使用 STATEMENT 格式的用户将无法迁移到 ROW 格式，主要原因有两点：一是 ROW 格式的二进制日志的大小要比 STATEMENT 大，二是在 ROW 格式中，用户或应用程序执行的 SQL 语句不会得到如实记录。为了解决这些问题，MySQL 5.6 与 MariaDB 10.0 添加了新功能。

8.5.1 ROW 格式的容量优化 MySQL

MySQL 服务器的二进制日志格式大致有 STATEMENT、ROW、MIXED 三种，其中最常用的是 STATEMENT 传统格式，但 STATEMENT 格式的二进制日志不能对 UUID 函数这类非确定型函数（NOT DETERMINISTIC）或功能进行复制。而 ROW 格式的二进制日志格式最终会将数据表中应用的列值记录到二进制日志，所以没有这样的限制。ROW 格式的二进制日志通常要比 STATEMENT 格式需要更多的存储空间。以上两种格式的优缺点十分明显，引入 MIXED 格式可以充分使用 STATEMENT 格式与 ROW 格式的优点。

但使用 MIXED 格式的二进制日志时，随着二进制日志大小的增加，仍然有可能需要更多的存储空间，或造成网络拥塞。MySQL 5.6 为了尽量减少 ROW 格式的二进制日志事件的大小而引入了 binlog_row_image 系统变量。binlog_row_image 系统变量可以设置为如下三个值之一。

FULL

与 MySQL 5.5 的 ROW 格式二进制日志一样，该方式与数据列的修改与否无关，修改记录的所有列都会记录到二进制日志。根据 INSERT、UPDATE、DELETE 语句的不同，记录到二进制日志文件的信息也不同。若为 INSERT 语句，则只将新插入记录的所有列（After-Image）记录到二进制日志文件；若为 UPDATE 语句，则修改前记录（Before-Image）与修改后记录（After-Image）的所有列都会记录到二进制日志；若为 DELETE 语句，则只将修改前记录（Before-Image）的数据列记录到二进制日志。

MINIMAL

MySQL 5.6 开始引入的新功能，将修改的内容应用到从服务器时，只将必需的数据列值记录到二进制日志。根据 INSERT、UPDATE、DELETE 语句的不同，记录到二进制日志中的内容也不同，详细内容将在后文讲解。

NOBLOB

该工作方式与使用 FULL 选项一样，但若 BLOB 数据列不发生改变，则不记录到二进制日志文件。

binlog_row_image 系统变量设置为 MINIMAL 时，根据 INSERT、UPDATE、DELETE 语句的不同，记录到二进制日志的内容如下所示。此处 PKE（Primary Key Equivalent）是指主键或起主键作用的数据列组合，对于不同数据表，PKE 的确定规则如下。

- 拥有主键的数据表：相应数据表的主键视为 PKE。
- 无主键时：相关数据表中若存在定义为 NOT NULL 的唯一索引，则该唯一索引视为 PKE；若不存在定义为 NOT NULL 的唯一索引，则记录的所有列组合视为 PKE。

	修改前的记录（Before Image）	修改后的记录（After Image）
INSERT	（无）	插入的记录中有明确值的数据列与 AutoIncrement 值（数据表中有 AutoIncrement 数据列时）
UPDATE	PKE	插入的记录中所有拥有明确值的数据列
DELETE	PKE	（无）

注意 MariaDB 尚不支持将 binlog_row_image 系统变量设置为 MINIMAL 或 NOBLOB 值。为了像 MySQL 5.6 一样减小 ROW 格式的二进制日志事件的大小，Monty Program AB 正在积极努力实现该功能。

8.5.2 ROW 格式二进制日志的信息性日志事件

目前仍有很多用户使用 STATEMENT 格式的二进制日志，之所以不使用 ROW 格式的二进制日志的主要原因是审计功能。使用 STATEMENT 格式时，用户或应用程序可以通过二进制日志文件查看 SQL 语句。但 ROW 格式中，由于只记录 SQL 语句修改的记录，所以无法查看准确的 SQL 语句。为了弥补这一不足，MariaDB 5.3 开始引入“注释事件”（Annotate row log event），MySQL 5.6 开始引入“信息性日志事件”（Informational log event）。

1. MariaDB 的注释事件

MariaDB 5.3 开始引入注释事件功能，利用该功能，在使用 ROW 格式二进制日志的情形下，将修改的记录信息写入二进制日志前，用户执行的 SQL 语句会先记录为 Annotate_ rows 二进制日志事件。若要启用注释事件功能，只需将 binlog-annotate-row-events 系统变量设置为 ON。

下列示例中，先在使用 ROW 格式二进制日志的 MariaDB 服务器内打开 binlog-annotate-row-events，执行几个简单的 SQL 后，再查看二进制日志的内容。

```
MariaDB> SHOW BINLOG EVENTS IN 'binary-log.000001';
+-----------------+---+-------------+---------+----------+-------------------------------+
|Log_name         |Pos|Event_type   |Server_id |End_log_pos|Info                          |
+-----------------+---+-------------+---------+----------+-------------------------------+
|binary-log.000001|4  |Format_desc  |100       |240        |Server ver: 5.5.24-MariaDB     |
|binary-log.000001|240|Query        |100       |331        |DROP DATABASE IF EXISTS test   |
|binary-log.000001|331|Query        |100       |414        |CREATE DATABASE test           |
|binary-log.000001|414|Query        |100       |499        |use 'test'; CREATE TABLE t1(a int) |
|binary-log.000001|499|Query        |100       |567        |BEGIN                          |
|binary-log.000001|567|Annotate_rows|100       |621        |INSERT INTO t1 VALUES (1), (2), (3)|
|binary-log.000001|621|Table_map    |100       |662        |table_id: 16 (test.t1)         |
...
```

上述示例中，名为 Annotate_rows 的事件即为注释事件。Annotate_rows 事件只起简单的注释作用，而不会在实际的从服务器中再次执行。此外，若在从 MariaDB 服务器中激活了 replicate- annotate-row-events 系统变量（ON），则获取自主服务器的注释（Annotate_rows）事件也会如实记录到从服务器的二进制日志。使用 MariaDB 的 mysqlbinlog 实用工具时，若二进制日志中有注释事件，则将其一起输出到画面，若不需要注释事件，则只使用 skip- annotate-row-events 选项即可。下列示例中，使用 mysqlbinlog 实用工具转储二进制日志时，输出注释事件。

```
shell> mysqlbinlog binary-log.000001
...
# at 1221
#100516 15:36:18 server id 100  end_log_pos 1289        Query   thread_id=2
     exec_time=0     error_code=0
SET TIMESTAMP=1274009778/*!*/;
BEGIN
/*!*/;
# at 1289
# at 1351
# at 1392
#100516 15:36:18 server id 100  end_log_pos 1351        Annotate_rows:
#Q> INSERT DELAYED INTO t3 VALUES (1), (2), (3)
```

```
#100516 15:36:18 server id 100  end_log_pos 1392        Table_map:
'test'.'t3' mapped to number 18
#100516 15:36:18 server id 100  end_log_pos 1426        Write_rows: table
id 18 flags: STMT_END_F

BINLOG '
stjvSxNkAAAAKQAAAHAFAAAAABIAAAAAAAAABHRlc3QAAnQzAAEDAAE=
stjvSxdkAAAAIgAAAJIFAAAQABIAAAAAAAAEAAf/+AQAAAA==
'/*!*/;
### INSERT INTO test.t3
### SET
###   @1=1 /* INT meta=0 nullable=1 is_null=0 */
# at 1426
# at 1467
#100516 15:36:18 server id 100  end_log_pos 1467        Table_map: 'test'.'t3'
 mapped to number 18
#100516 15:36:18 server id 100  end_log_pos 1506        Write_rows: table
id 18 flags: STMT_END_F
```

2. MySQL 5.6 的信息性日志事件

与 MariaDB 一样，MySQL 5.6 也引入了注释事件功能。MySQL 5.6 将该功能称为“信息性日志事件”，通过 binlog-rows-query-log-events 系统变量决定是否启用该功能。与 MariaDB 类似，在 ROW 事件之前也会先输出 SQL 语句，Event_type 标识为 Rows-query。

```
mysql> SHOW BINLOG EVENTS IN 'binary-log.000001';
+----------------+-----+------------+---+---+-------------------------------------+
| Log_name       | Pos |Event_type  |Server_id|End_log_pos|Info                     |
+----------------+-----+------------+---+---+-------------------------------------+
|binary-log.000001|   4 | Format_desc |100 |114 | Server ver: 5.6.15-log, Binlog ver: 4 |
|binary-log.000001| 114 | Query       |100 |200 | use 'test'; CREATE TABLE NUM_INTS (a INT) |
|binary-log.000001| 200 | Query       |100 |268 | BEGIN                                |
|binary-log.000001| 268 | Rows_query  |100 |323 | # INSERT INTO NUM_INTS VALUES (1), (2), (3) |
|binary-log.000001| 323 | Table_map   |100 |364 | table_id: 54 (test.NUM_INTS)         |
|binary-log.000001| 364 | Write_rows  |100 |408 | table_id: 54 flags: STMT_END_F       |
|binary-log.000001| 408 | Query       |100 |477 | COMMIT                               |
...
```

MySQL 5.6 使用 mysqlbinlog 实用工具将二进制日志内容变换为 SQL 文件时，信息性日志事件也会一起显示到画面。使用 MySQL 5.6 的 mysqlbinlog 实用工具时使用 -vv 选项，可以同时查看 SQL 语句。

```
shell> mysqlbinlog -vv binary-log.000001
```

```
...
# at 268
#110401 14:24:29 server id 1  end_log_pos 323   Rows_query
# INSERT INTO t1 VALUES (1), (2), (3)
# at 323
#110401 14:24:29 server id 1  end_log_pos 364   Table_map: 'test'.'NUM_INTS'
 mapped to number 54
# at 364
#110401 14:24:29 server id 1  end_log_pos 408   Write_rows: table id 54
flags:STMT_END_F
```

8.6 延迟复制 MySQL

MySQL 与 MariaDB 服务器复制的本来目的是以最快的速度保持同步，使主从服务器间的数据保持一致。也就是说，主从服务器之间越没有延迟，使用从服务器进行故障恢复（Fail-Over）就越容易，这在使用服务型从服务器时是必需的。但是，有时要特意延迟主从服务器间的复制。假设开发者或管理员因操作失误而删除（DROP 或 DELETE）了主 MySQL 服务器中的重要数据表。发现失误时，从服务器也已经执行了删除查询。此时只能停止服务，使用备份恢复删除的数据表及其数据。

为了灵活应对这种情形，MySQL 5.6 添加了延迟复制功能，许多第三方软件也提供了该功能。使用 MySQL 5.6 的延迟复制功能时，如果使用 CHANGE MASTER 命令进行连接复制，则只要设置延迟多长时间再执行复制即可。下列示例中，执行 CHANGE MASTER 命令时使用 master_delay 选项，并将其设置为 3600 秒。这样，从服务器的数据改动要比主服务器晚 1 个小时。

```
MySQL> CHANGE MASTER TO master_host='',master_port=3308,
                        master_user='',master_password='',
                        master_log_file='binary-log.000019',master_log_pos=120,
                        master_delay=3600;
```

启用延迟复制后，执行 SHOW SLAVE STATUS 命令。从命令的执行结果可以得知延迟执行的时间以及到下一个事件执行时的剩余时间。

```
MySQL> SHOW SLAVE STATUS\G
*************************** 1. row ***************************
               Slave_IO_State: Waiting for master to send event
                  Master_Host: 127.0.0.1
                  Master_User: root
                  Master_Port: 3308
```

```
                    ...
      Seconds_Behind_Master: 0
                    ...
                SQL_Delay: 3600
      SQL_Remaining_Delay: 3555
    Slave_SQL_Running_State: Waiting until MASTER_DELAY seconds after master
executed event
                    ...
```

从 SHOW SLAVE STATUS 命令的执行结果可以看到，SQLDelay 为 3600 秒，到执行下一个事件还剩余 3555 秒。像这样延迟处理复制时，只要延迟不超过 3600 秒，表示从服务器延迟的 Seconds_Behind Master 列值就显示为 0。即使使用延迟复制功能，主服务器的二进制日志也会立即复制到从服务器的中继日志文件，只延迟执行 SQL 线程。因此，主服务器因故障导致磁盘数据无法恢复时，可以使用从服务器的中继日志进行恢复，就像不使用延迟复制一样，通过从服务器设备完成故障恢复。

8.7 MariaDB 与 MySQL 服务器间的复制

使用 MySQL 的状态下，升级到 MySQL 5.6 或者迁移到 MariaDB 时，若想在不停止服务的前提下进行实时迁移，MariaDB 服务器的复制功能是必需的。下表列出了各版本之间复制功能的兼容情况，这些结果都经过了简单的查询测试。

主服务器 / 从服务器	MariaDB 5.5	MariaDB 10.0	MySQL 5.5	MySQL 5.6
MariaDB 5.5	O	O	O（MicroSecond 功能不可用）	O（MicroSecond 功能不可用）
MariaDB 10.0		O		O（GTID 可用）
MySQL 5.5	O	O	O	O
MySQL 5.6		X		O

MySQL 或 MariaDB 主服务器比从服务器版本更先进（新发布的版本）时，通常都不正式支持复制。上表加粗部分就属于这种情况。若主服务器为 MySQL 5.6，则仅当从服务器也为 5.6 版本时，才能正常复制。若主服务器为 MySQL 5.6，而从服务器不是 MySQL 5.6，那么从服务器就会因为无法理解主服务器的二进制日志而发生如下错误。从错误内容看好像是二进制日志文件的位置设置错误，其实是二进制日志格式不同引起的。

```
Got fatal error 1236 from master when reading data from binary log: 'Client
requested mas
ter to start replication from impossible position'
```

此外，从 MySQL 5.5 或 MariaDB 5.5 升级到 MySQL 5.6 或 MariaDB 10.0 好像没有什么特殊

问题。但从服务器由 MySQL 5.5 或 MySQL 5.6 替换为 MariaDB 5.5 时，主 MariaDB 服务器将无法使用以微秒与毫秒为单位的 DATETIME 与 TIME 类型。

为了最大限度地保持 MariaDB 与 MySQL 之间二进制格式的兼容性，主服务器为 MariaDB 而从服务器为 MySQL 时，MariaDB 不会将非兼容但可忽略的二进制日志传送给从服务器。此外，主服务器为 MySQL 而从服务器为 MariaDB 时，MariaDB 会尽最大努力接收不必要的二进制日志事件而不发生错误。但这种方式下得到的兼容性相当有限。若一定要在 MariaDB 与 MySQL 服务器之间进行复制，则需要进行长时间测试，并使用多个实例进行兼容性检查。

注意 上述表格中，复制功能可用与否是使用基本的 DML 语句（INSERT、UPDATE、DELETE 语句与 DDL 命令）测试得到的。实际的服务环境中，各种形式的 SQL 与 DML 语句是否能够 100% 无问题运行，则必须根据相应的服务环境构建测试系统并进行严格测试。

此外，MariaDB 10.0 尚无法理解 MySQL 5.6 的二进制日志格式。但相信随着 MariaDB 服务器性能的稳定及其功能的进一步改善，MariaDB 10.0 很快就能理解 MySQL 5.6 的二进制日志格式。

8.8 其他功能改善 MariaDB MySQL

除前面讲解的几大功能外，MariaDB 10.0 与 MySQL 5.6 还对许多其他功能进行了细小改善。下面只讲解几种必知的功能改善。

8.8.1 二进制日志校验和

MariaDB 5.3 开始添加了二进制日志校验和功能，下面三个系统变量可以决定是否启用 MariaDB 的二进制日志校验和功能。

binlog_checksum

binlog_checksum 系统变量用于决定记录二进制日志时是否针对各事件计算校验和并将其写入日志，以及使用哪种算法。binlog_checksum 系统变量为布尔类型，可以将其设置为 0（ON）或 1（OFF）。

master_verify_checksum

从服务器从主服务器读取二进制日志时，主服务器的二进制日志转储线程读取二进制日志并交给从服务器。此时，master_verify_checksum 系统变量决定主服务器的二进制日志转储线程是否检查二进制日志的校验和。master_verify_checksum 系统变量为布尔类型，可以将其设置为 0（ON）或 1（OFF）。

slave_sql_verify_checksum

该系统变量决定从服务器的 SQL 线程是否计算来自主服务器的二进制日志的校验和，并检查二进制日志是否发生损坏。slave_sql_verify_checksum 系统变量为布尔类型，可以将其设置为 0（ON）或 1（OFF）。

为了检查二进制日志是否发生损坏，MySQL 5.6 也引入了二进制日志校验和功能。通过下

面三个系统变量可以启用或关闭二进制日志校验和功能。

binlog_checksum

binlog_checksum 系统变量决定记录二进制日志时是否针对各事件计算校验和并将其写入日志，以及使用哪种算法。与 MariaDB 不同，MySQL 5.6 可以从 CRC32 与 NONE 之中选用一种。若设置为 NONE，则表示不生成二进制日志校验和；若设置为 CRC32，则表示使用 ISO-3309 CRC32 算法计算校验和并将其写入二进制日志。

master_verify_checksum

该系统变量的功能与 MariaDB 的 master_verify_checksum 系统变量相同，也为布尔类型。

slave_sql_verify_checksum

该系统变量的功能与 MariaDB 的 slave_sql_verify_checksum 系统变量相同，也为布尔类型。

从服务器的 SQL 线程计算二进制日志校验和并进行验证时，若出现问题，则会引发如下错误。

```
MySQL_Master> SHOW BINLOG EVENTS FROM 198232;
ERROR 1220 (HY000): Error when executing command SHOW BINLOG EVENTS: Wrong
offset or I/O error

MySQL_Slave> SHOW SLAVE STATUS\G
...
Last_SQL_Errno: 1594
Last_SQL_Error: Relay log read failure: Could not parse relay log event
entry. The possible reasons are: the master's binary log is corrupted...
...
```

试图从主 MySQL 服务器读取损坏的二进制日志时，就会发生“..Wrong offset or I/O error”错误，只显示文件读取失败信息而不输出校验和错误信息。损坏的二进制日志传递到从服务器后，从服务器中会发生“the master's binary log is corrupted”错误并停止复制。

8.8.2 二进制日志 API

MySQL 5.6 开始提供二进制日志 API，在 C/C++ 程序中调用这些 API 可以直接连接 MySQL 服务器，以分析磁盘存储的二进制日志文件。开发应用程序时，借助这些二进制日志 API 可以直接连接 MySQL 服务器获取二进制日志，将数据传递给 HBase 或 Cassandra 这类 NoSQL 数据库，也可以将存储在 MySQL 服务器的数据自动保存（SET）到 Memcached 或 Redis 这类缓存服务器。

也就是说，通过二进制日志 API，可以使用 Oracle 数据库的 CDC（Change Data Capture）功能。其实，MySQL 中也将其称为 CDC。关于二进制日志 API 的详细内容，请参考 MySQL 研究组（MySQL Search Team）的博客（http://intuitive-search.blogspot.kr/）。

8.8.3 提交二进制日志组

MySQL 5.5 及其以前版本中，将 sync_binlog 选项设置为 1 后，每当事务发生，就会将二进制日志写入磁盘并刷新。图 8-13 中，每次提交事务时，会先记录二进制日志并刷新，然后再提交给存储引擎。此时，MySQL 服务器中要经历 Prepare 与 Commit 两个阶段，这称为“两阶段提交”（XA，Two-Phase Commit）。MySQL 服务器中，由于二进制日志的工作方式与存储引擎类似，所以存储引擎与二进制日志采用两阶段提交方式进行同步。MySQL 中，存储引擎与二进制日志文件之间的两阶段提交又称“内部两阶段提交”（Internal two phase commit）。

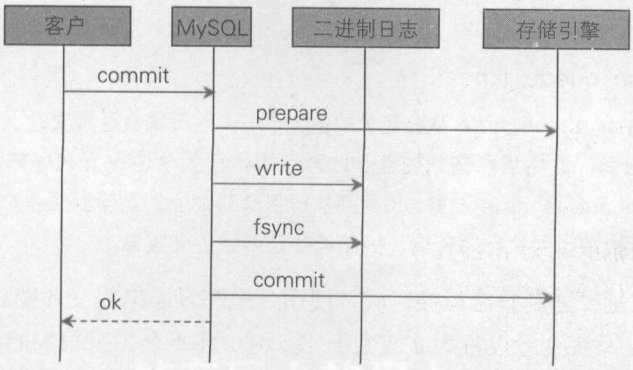


图 8-13 事务提交与二进制日志刷新

但若 sync_binlog 选项设置为 1，写磁盘负荷就会相当重，读写数据文件可能变慢。MySQL 5.6 引入了二进制日志组提交功能，使用该功能时，每当有事务提交，二进制日志仍会写入磁盘，但会收集多个事务的二进制日志一次性写入磁盘并刷新。图 8-14 表示 MySQL 5.6 中的事务提交与二进制日志组提交的工作方式。

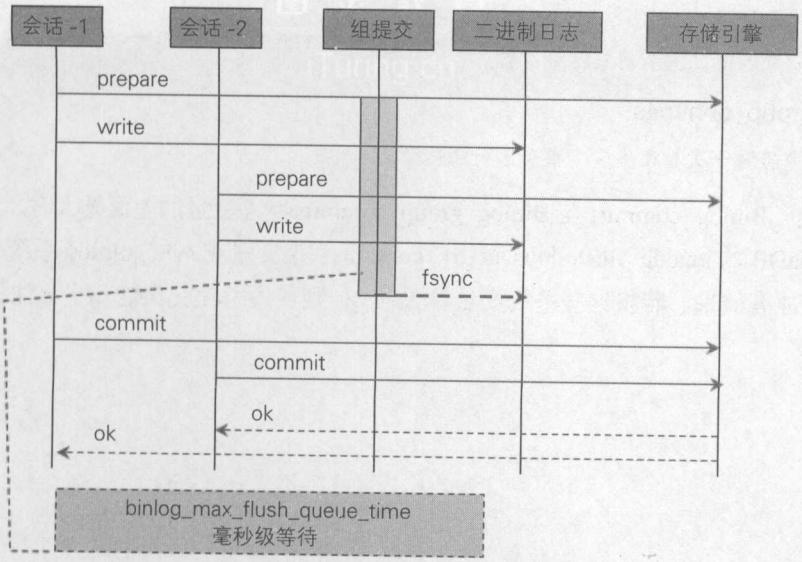


图 8-14 组提交的工作方式

二进制日志组提交自动处于激活状态，即使不支持组提交的存储引擎，MySQL 引擎级别也会将二进制日志放入组并进行组提交。InnoDB 存储引擎提供如下两个选项，以进行调优。

binlog_order_commits={0|1}

binlog_order_commits 系统变量为布尔型，可以设置为 0（ON）或 1（OFF）。该系统变量设置为 1 时，将按照事件记录到二进制日志文件的顺序进行提交；设置为 0 时，将并行处理事务，事务多时会表现出一定的性能提升。当然，由于在相同时间点提交事务，同时修改一条记录的事务几乎不可能进行组提交，所以没有必要同时获取二进制日志的顺序与提交的顺序。即使 InnoDB 提交与二进制日志的记录顺序不同，若无特殊问题，仍可以将 binlog_order_commits 系统变量设置为 0，以略微提升系统性能。

binlog_max_flush_queue_time

若 binlog_max_flush_queue_time 系统变量的值大于 0，则每次执行提交时，二进制日志会先在刷新队列中等待一定时间，然后再刷新到磁盘。此时可以微秒为单位设置在刷新队列中等待的时间。若 binlog_max_flush_queue_time 设置为 0，则执行提交时会立即刷新到磁盘。但即使 MySQL 服务器中连接数很多，且频繁实时执行短查询，仍然能够获得组提交效果。

只有同时发生大量二进制日志写操作时，使用二进制日志组提交才能获得明显的性能提升。因此，将 sync_binlog 系统变量设置为 0 或大于 4~5 时，将不会带来很大的性能提升。

MariaDB 从 5.3 版本开始支持组提交功能，MariaDB 的组提交功能既可应用于 InnoDB 的日志文件，也可应用于 MariaDB 的二进制日志文件。组提交方式类似于 MySQL 5.6，允许多个线程同时向磁盘写入二进制日志，收集之后再统一执行一次刷新操作（fsync 或 fdatasync 系统调用）。MariaDB 中，组提交功能处于自动激活状态，无需设置任何系统变量，通过状态变量可以查看二进制日志组提交功能的执行效率。

Binlog_commits

保存事务提交时的二进制日志记录次数。

Binlog_group_commits

指定提交事务时一次收集多少个事务进行组提交。

可以断定，Binlog_commits 与 Binlog_group_commits 状态变量的差值越大时，组提交的效果越显著。MariaDB 的 innodb_flush_logs_at_trx_commit 系统变量或 sync_binlog 系统变量只有设置为 1，组提交才有效果。若将这些系统变量设置为 0，则使用组提交功能几乎无法提升性能。

关注图灵教育 关注图灵社区

iTuring.cn

在线出版　电子书《码农》杂志　图灵访谈 ……



QQ联系我们

读者QQ群：218139230　164939616



微博联系我们

官方账号：@图灵教育 @图灵社区 @图灵新知

市场合作：@图灵袁野

写作本版书：@图灵小花 @图灵张霞

翻译英文书：@朱巍ituring @楼伟珊

翻译日文书或文章：@图灵乐馨

翻译韩文书：@图灵陈曦

电子书合作：@hi_jeanne

图灵访谈/《码农》杂志：@李盼ituring

加入我们：@王子是好人



微信联系我们



图灵教育
turingbooks



图灵访谈
ituring_interview

一本书搞定
MariaDB 10.0与MySQL 5.6!



本书讲解了MariaDB的诞生背景及设计目的，并通过与MySQL的比较使读者轻松理解MariaDB。本书内容涉及MariaDB 5.1~10.0的各个方面，讲解重点在于MariaDB 10.0的各种实用功能。此外，MariaDB与MySQL有着紧密的联系，书中对MariaDB 10.0与MySQL 5.6的新功能进行了比较，读者可以同时学习MariaDB与MySQL的内容。

本书涵盖内容

- MariaDB与MySQL间的兼容性
- MariaDB的安装与迁移
- 在线修改Schema
- 执行计划分析
- MariaDB 10.0与MySQL 5.6的优化
- MariaDB的存储引擎
- MariaDB的附加功能（线程池及管理诊断工具）
- XtraBackup备份工具
- MariaDB 10与MySQL 5.6的复制（GTID、多源复制及多线程从服务器）

* 若对本书所讲内容有疑问，特别是有关MariaDB与MySQL技术方面的问题，请在作者博客（http://cafe.naver.com/realmysql）留言。

위키북스

图灵社区：iTuring.cn
热线：(010) 51095186 转 600

分类建议 计算机/数据库

人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-40908-9



ISBN 978-7-115-40908-9

定价：89.00元